PROMETHEUS AI
Phase 2

Sean Stappas
260639512

ECSE-499: Honours Thesis II

Supervised by: Prof. Joseph Vybihal

December 7th, 2017

Abstract

Prometheus AI is a model of the human brain aimed at controlling multiple robots to achieve some goal. Possible application areas include environments hazardous for humans, such as in 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 recommendations for actions to the META. The META represents high-level thinking and makes an intelligent decision to either accept the recommendations from the ES or to initiate a new thinking cycle in the KNN. The tasks for this thesis were to implement the KNN and ES layers in Java and to supervise work done by undergraduate students on Prometheus. This was achieved using design criteria such as efficiency, readability and testability. Continuous integration tests were created on GitHub[1] with TestNG and TravisCI and extensive documentation was written in Javadoc[2].

Acknowledgments

Under my supervision, several undergraduate students worked on the Prometheus layers in Prof. Vybihal's lab. Over the summer, Isaac Sultan and Si Yi Li made contributions to the ES and KNN layers, respectively. This semester, Mohammad Owais Kerney and Michael Ding showed interest in the NN and META layers, respectively.

[1]The GitHub repository can be found here:
`https://github.com/seanstappas/prometheus-ai/`

[2]The Javadoc can be found here:
`http://seanstappas.me/prometheus-ai/`

Contents

List of Figures

List of Tables

Abbreviations

The following abbreviations will be used throughout the report:

Table 1: Abbreviations in Prometheus.

Abbreviation	Meaning
NN	Neural Network
KNN	Knowledge Node Network
KN	Knowledge Node
ES	Expert System
META	Meta Reasoner

1 Introduction

Prometheus is an artificial intelligence system designed to control multiple robots to achieve some goal. All the thinking and decision-making is done by the system, which collects data from all the robots' sensors.

Applications for this type of system include robots in hazardous environments, such as in outer space (Mars, Moon, etc.), in nuclear plants after a disaster and in military zones. The system could theoretically learn from this data in a given environment and apply its learning to new environments.

The architecture of Prometheus is loosely inspired from the structure of the human brain and is composed of the following four layers [1]: the Neural Network (NN), the Knowledge Node Network (KNN), the Expert System (ES) and the Meta Reasoner (META). A high-level diagram of the system can be seen in Figure 1.

The scope of this thesis is to implement the KNN and ES layers of Prometheus. Therefore, more detail will be given for these layers.

The theory needed to understand these four layers will be given in Section 2. The assigned task will then be described in Section 3. Design criteria will be discussed in Section 4. The description of the work done can be found in Section 5. The results and tests will be discussed in Section 6. Finally, the possible future impact of this project on society and the environment will be explored in Section 7.

2 Background

This section draws heavily from the background discussed in the previous report [2].

2.1 Neural Network

The NN layer consists of a network of neurons with a structure similar to neurons in the human brain. It is the interface between the robots' sensors and the rest of the AI system.

The robots in Prof. Vybihal's lab are equipped with two types of sensors: camera and ultrasonic. The ultrasonic sensor can measure distance between the robot and nearby objects and the camera can take images of what the robot is facing.

The NN gathers raw sensor data and will build an abstract view of the robot's surrounding environment. For each camera image, it will achieve two main goals:

1. Classify objects observed in the image.
2. Localize objects in the image.

Ultimately, the classification and localization of objects will produce abstract informational tags, which will be passed on to the KNN. These tags can therefore be characteristics of objects in the world, such as distance. The tags can also have associated confidence values, which can be converted to belief within the KNN.

2.2 Knowledge Node Network

The KNN layer is the analog to memory in the human brain. It takes in the tags provided by the NN and outputs tags based on its knowledge. It consists of interconnected Knowledge Nodes (KNs), which are abstract structures representing memories and their connections to other memories. A simple model of a Knowledge Node (KN) can be seen in Figure 2.

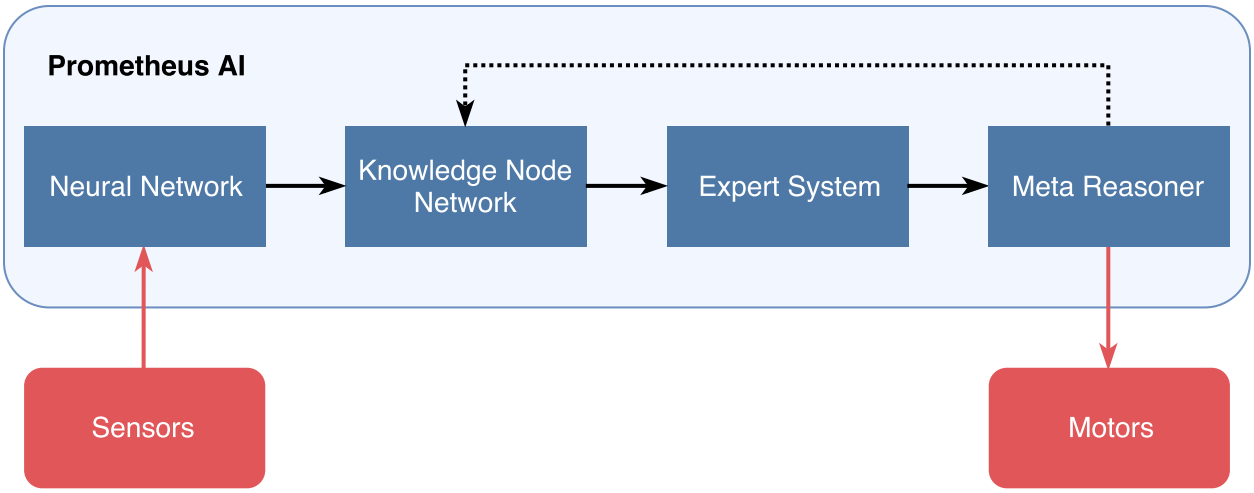


Figure 1: Prometheus AI model. Blue represents Prometheus and red represents the robots.

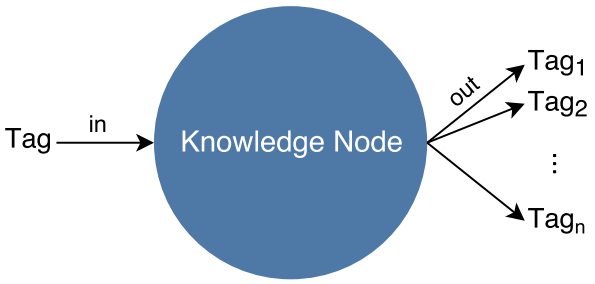


Figure 2: High-level model of the Knowledge Node (KN) of the KNN.

2.2.1 Excitation and Firing

KNs have an input tag representing some datum and output tags representing related data. When a KN is stimulated it becomes "excited". Tags in the KNN can become "active", potentially exciting the associated KN. We say that the associated KN "fires" if it is excited beyond a threshold. For example, if the NN observes a ball and provides that information to the KNN, then a "ball" tag may become active in the KNN. This would excite the KN with input tag "ball". If that KN fires, other tags related to that observation can become active. For example, tags representing ball characteristics such as "round object" may become active. This tag in turn may be connected to another KN, potentially causing more activation. More details about the structure of these tags will be provided in Section 2.3, where the Expert System will be discussed.

2.2.2 Strength

There is a strength value associated with every KN which represents how much weight an activation has and therefore how quickly that node will fire. Strength can be seen as the firing predisposition of neurons in the brain as a result of learning [3]. Indeed, learning can increase the synaptic strength between neurons and cause early firing of those neurons [4]. For example, a person who has had a bad experience with spiders would fire their fear response upon seeing a spider more quickly than one without that fear.

2.2.3 Belief

Every KN can have a belief value associated with it, representing how certain the KNN is that the input tag is true. In this case, the belief value would be stored within the KN itself. These belief values can come from the NN, since a neural network can associate confidence values with its classifications of objects [5]. When initiating a think cycle, the belief values at each stage can be multiplied with each other to produce a new belief value, making the KNN less certain of a memory as it searches through its tree of KNs. This represents how belief changes when thinking. Indeed, some memories in the brain require a great deal of thinking to reach and, as such,

can be less certain than other memories. This can lead to the recollection of false memories [6].

Note that the activation of a KN and the belief value associated with it are two different concepts. Indeed, a KN may be fired even if the KNN knows that it is false. This can represent an agent "lying" to itself or repeating a thought process it knows to be false. This is known as "self-deception" in humans and it has been argued that this can be used for positive mental health [7]. In the context of Prometheus, this may help the system achieve its desired goal.

2.2.4 Search

Searching in the KNN represents a tag activation routine. The KNN has four ways of searching: direct, forward, backward and lambda. Thinking in the KNN is a related concept to searching. When thinking, the KNN uses its currently active tags to activate further tags. In this way, thinking is a special case of searching, where the input tags of the search are the active tags in the KNN. The KNN has three ways of thinking: forward, backward and lambda. The version of searching or thinking to be done by the KNN is chosen by the META.

Direct searching is a simple lookup of the desired tag and excitation of the associated KN. This is the simplest search in the KNN and can be seen in Figure 3.

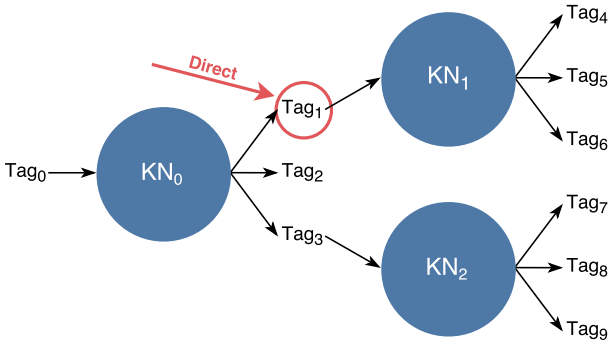


Figure 3: Direct searching in the KNN.

Forward searching or thinking is depicted in Figure 4. Firing a KN can cause forward activation of more KNs, hence the "forward" naming. It repeatedly performs the same task as direct search on activated tags.

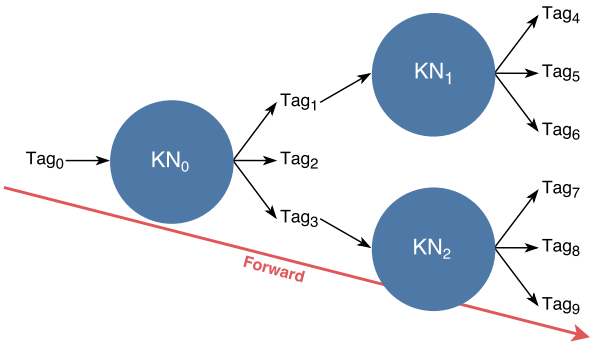


Figure 4: Forward searching or thinking in the KNN.

Searching or thinking backward starts at the output tags of KNs and works backwards, as can seen in Figure 5. It checks the output tags of a KN and, if a certain ratio of them match the search input tags, the KN's input tag is activated. This ratio is known as the partial match ratio.

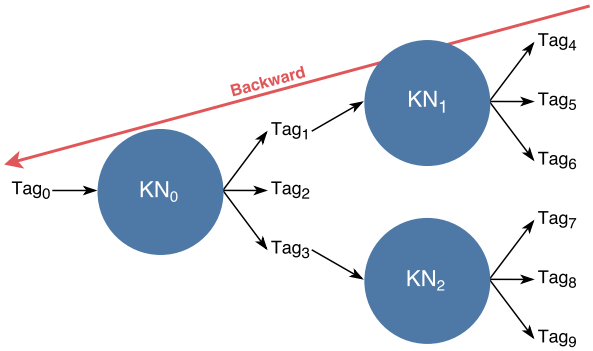


Figure 5: Backward searching or thinking in the KNN.

The partial match ratio relates to the extent to 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 will often classify what they observe with some uncertainty [8] and this can be represented with backwards thinking. This type of thinking occurs constantly in the background in humans [3].

Lambda searching or thinking uses a combination of forward and backward. It can can be seen in Figure 6. It is called "lambda" because the shape of the thinking trajectory resembles a Greek uppercase lambda (Λ). It first looks at output tags of KNs and propagates activation backwards, similar to backward search. After a certain amount of nodes are fired, it will then start activating forwards, similar to forward search. 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.

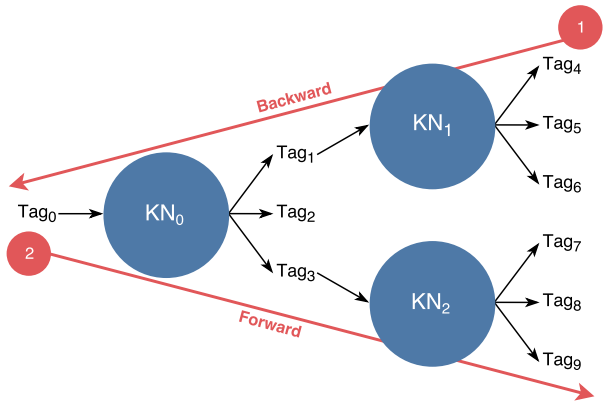


Figure 6: Lambda searching or thinking in the KNN.

Lambda thinking occurs in humans when using analogical reasoning to find a memory [9]. In essence, when a person wants to locate a memory that is not directly accessible, they will explore related memories, move their way "backwards" to similar concepts and think "forwards" to focus in on the desired memory. For example, if one is asked where they were in 2002 on a specific date, they most likely would not remember. If they explore related dates and events in their life, however, they may be able to narrow down their thoughts and extract that memory. In the context of the Prometheus system, lambda search will be attempted if all other searches fail (forward and backward).

All forms of searching or thinking can continue until there are no more tags to activate, which corresponds to natural quiescence. There can also be a fixed number of thinking cycles, which represents how much effort is being put into thinking. Indeed, in humans, thinking is done with varying degrees of effort [10]. This number of thinking cycles also represents how deep the search will go into the KN graph and is therefore called the "ply" of the search.

2.2.5 Aging

Age represents how long it has been since a KN has been excited. After a KN has aged a certain amount of time, that node will be discarded, similarly to how old memories are discarded in the brain [11]. In backward search, "young" KNs (with a low age) are visited first. Here, there can also be an age threshold such that the KNN only searches through KNs below a certain age.

The result of all forms of searching or thinking is a collection of activated tags, which are passed on to the ES.

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.

2.3.1 Facts

Facts are simple calculus predicates showing that something is true. Their form can be seen in Equation (1), where P is the predicate name and each A_i is an argument.

$$Fact \coloneqq P(A_1, \ldots, A_n) \tag{1}$$

Table 2 shows some example facts with simple arguments and their meanings. Note that the arguments are relevant when matching with other facts.

As a concrete example, a fact argument can represent a certain measurement, e.g., $distance = 5$ representing a robot's distance from a wall as measured by one of its sensors.

Table 2: Examples of facts in the ES.

Fact	Meaning
$P(x)$	x is true or active.
$P(x = 1)$	x is equal to 1.
$P(x\ != 1)$	x is not equal to 1.
$P(x > 1)$	x is greater than 1.
$P(x < 1)$	x is less than 1.
$P(\&x)$	x can take any integer value.
$P(*)$	All arguments can take any value.
$P(?)$	One argument can take any value.

2.3.2 Recommendations

Recommendations represent suggestions for actions to be taken by a robot. They take on the same predicate form as Facts, as shown in Equation (2). The only difference is that it is customary to prefix the predicate name with a '@' character. 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 perform that action.

$$Recommendation \coloneqq @P(A_1, \ldots, A_n) \quad (2)$$

2.3.3 Rules

Rules are many-to-many structures with facts as inputs and tags as outputs. This can be seen in Equation (3), where $m \geq 1$ and $n \geq 1$, i.e., there must be at least one input fact and one output tag. Each output tag can either be a fact or a recommendation. When all the input facts become active, the output tags become active and the rule itself is said to be active. In this way, a rule can represent a logical AND of all its input facts.

$$Rule \coloneqq Fact_1 \ldots Fact_m \rightarrow Tag_1 \ldots Tag_n \quad (3)$$

2.3.4 Thinking

The runtime of the ES consists of the following general steps [12]:

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 represents the activation routine of the ES. This 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 number of thinking cycles, which represents the effort put into thinking, similarly to the KNN. The recommendations activated as a result of thinking are passed on to the final layer, the META.

2.4 Meta Reasoner

The META layer represents high-level reasoning in the human brain. It is aware of its environment and context. It makes decisions based on what it believes to be right. It is paranoid and constantly checks whether the recommendations suggested by the ES 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 recommendations from the ES, it may send a query back to the KNN to initiate another think cycle and generate new recommendations, as can be seen in Figure 7.

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.

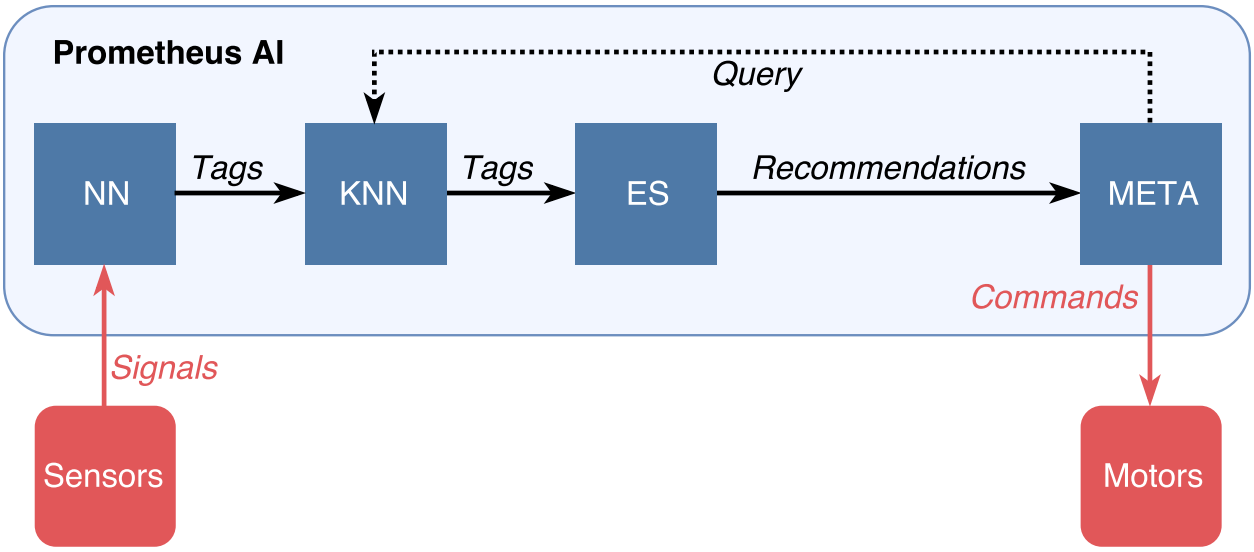


Figure 7: Prometheus AI model with labeled input and output.

3 Problem

The main task for this thesis is to construct the KNN and ES in Java. The requirement is that these layers should implement the functionality described in Section 2. Another important task is to supervise the work of other undergraduate students working on Prometheus and to put systems in place to ensure a certain standard for the code. The design criteria for both these tasks will be discussed in Section 4 and their implementation will be discussed in Section 5.

4 Design

4.1 Efficiency

A very important consideration when designing the system is speed. Since the robots may have to react very quickly to stimuli 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.

4.2 Object Oriented Programming

Another important design choice is to leverage object-oriented programming (OOP) as much as possible. OOP allows code to be very clean and reusable [13]. Principles such as polymorphism, encapsulation and abstraction will be followed closely. Indeed, the system should be as abstract as possible, while still performing its desired task. For instance, the system should be general enough to perform under simulations as well as in real-life environments. It should also ideally be able to perform in vastly different real-life environments, with varying tasks. Encapsulation can also be very useful, since each layer of the system has unique, localized functionality that does not need to be visible from the rest of the system.

The OOP principles of SOLID [14] will also be followed:

Single responsibility principle: a class should only have a single responsibility. In particular, a class should only have one reason to change.

Open/closed principle: a software entity's functionality can be extended without changing its source code.

Liskov substitution principle: a class instance can be replaced by an instance of a subtype of that class while remaining correct.

Interface segregation principle: a user of an

interface should not be forced to depend on unnecessary methods. This encourages the use of multiple client-specific interfaces.

Dependency inversion principle: high-level and low-level modules should depend on abstractions. In particular, a high-level module does not need to know the details of implementation of a low-level module and vice-versa.

4.3 Readability and Documentation

The code written should be very easy to understand. This means implementing each method and class in the most intuitive way possible and providing good documentation to support the code. This is to ensure that anyone wanting to work with the code or looking to understand how the system works has an easy time doing so.

4.4 Testing

An important criterion is that the code be heavily tested. This includes integration tests, covering end-to-end functionality, and unit tests, covering specific methods or classes. The unit tests should be behavior-driven, so that a developer reading the tests can easily understand the purpose of the element being tested.

4.5 Quality Assurance

With the numerous undergraduate students showing interest in working on Prometheus, there must be a system in place to ensure the quality of the code is upheld. This may include automatic testing of the code, as well as human review of code before being committed.

5 Implementation

5.1 Libraries

Prometheus is a Java Maven project. This allows all the library dependencies to be specified in a `pom.xml` file in the root of the project directory. Maven allows dependencies to be easily added or removed in a Java project without needing to keep track of `jar` files.

An important library that was used is Google Guice. This is the backbone for all the dependencies in the code. Guice neatly allows the implementation of various important OOP principles, like dependency inversion. Notably, the `es` and `knn` packages have associated Guice modules which are used by the high-level Prometheus module. Guice factories were also created for classes which require special constructor arguments. A diagram of the various Guice dependencies in the project can be seen in Figure A.1 in Appendix A.

The GraphStream library was also used for graph visualization and will be discussed in Section 5.8. The Apache Commons Lang library was used for standardized implementations of the `hashCode()`, `equals()`, `compareTo()` and `toString()` methods of Java objects. The very important TestNG and Mockito libraries were used for testing and will be discussed in Section 6.

It is also important to note that Promtheus uses Java 8 because of the numerous improvements it introduced, including `Optional` objects and lambda expressions.

5.2 Directory Structure

Below is a description of the contents of each top-level directory in the project.

`data`	Input data files for the KNN.
`docs`	Javadoc files.
`graphs`	Graphs created by the graphing tools.
`reports`	Reports on Prometheus.
`src`	Source code.

The `src` directory then contains the `main/java/` directory containing the main Prometheus code and the `test/java/` directory containing tests.

5.3 Package Structure

Within the main source code directory, the `prometheus`, `tags`, `nn`, `knn`, `es`, `meta` and `graphing` Java packages were created. Note that the `nn` and `meta` packages are simple skeletons of the NN and META layers. For most of these packages, sub-packages named `api`, `guice` and `internal` were created. Below is a description of the contents each of these packages should have.

`api` Public classes and interfaces. Only code relevant for a user of the package should be present here.

`guice` Public Guice module. This module will be used by a user of the package and should install an internal Guice module.

`internal` Internal classes and interfaces. Internal code that does not concern a user is found here, as well as an internal Guice module to install internal classes.

5.4 Prometheus Interface

A high-level `Prometheus` interface was created and was placed in the `prometheus` package. Its dependencies can be seen in Figure 8. The use of the `NeuralNetwork`, `KnowledgeNodeNetwork`, `ExpertSystem` and `MetaReasoner` interfaces is a good example of many of the SOLID principles, like single responsibility, interfaces segregation and dependency inversion. This interface allows a user to access the NN, KNN, ES and META layers via getter methods.

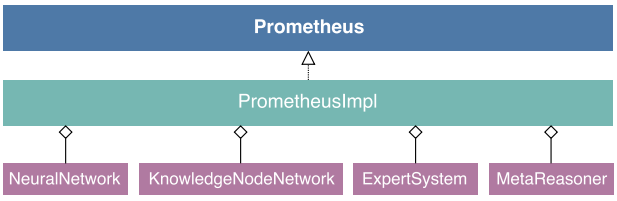


Figure 8: UML diagram of the `Prometheus` dependencies.

5.5 Tags

As described in Section 2, the entire system revolves around tags passed from layer to layer. For this reason, careful thought was put into the design of these tags.

The tags need to be as general as possible. A natural choice for this structure would be a `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. Furthermore, if specific functionality is needed in the future for each tag type, that can easily be implemented with a Java class.

For these reasons, the tags are implemented using an abstract `Tag` class, with `Predicate` and `Rule` subclasses. The `Predicate` abstract class is a superclass of the `Fact` and `Recommendation` classes, as shown in Figure 9. This is a good example of using the inheritance principle of OOP. The Liskov substitution principle was also kept in mind when designing these subclasses. The class strategy should also make manipulating the Tags faster, while incurring a slight memory overhead. To store these Tags in a database, they could be converted to JSON format using a library like Google Gson. On read from the database, they could easily be decoded using the same library. All these classes are present in the `tags` package of the code.

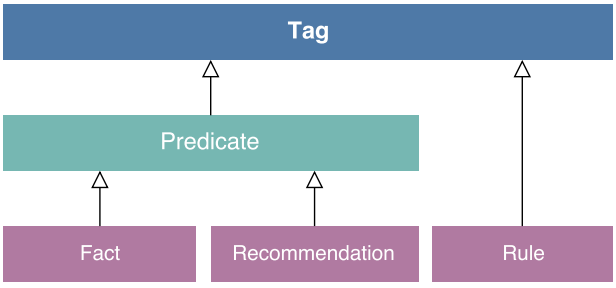


Figure 9: UML diagram of the major classes in the `tags` package.

The `Tag` and `Predicate` classes themselves are abstract. This means that an object may not be directly instantiated as either, 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. This is an example of using the abstraction principle of OOP.

The `Rule` class has the following important fields:

`inputFacts` `Set` of input `Facts`.

`outputPredicates` `Set` of output `Predicates`.

The `Fact` and `Recommendation` classes have the following fields:

`predicateName` `String` representing the predicate name.

`arguments` `List` of arguments to the predicate.

It is also very important that these tag objects be immutable by a user after creation, since they will be used as keys to map to KNs in the KNN, as will be discussed in the next section. To achieve this, the objects may have getter methods, but do not have any setter methods. In addition, if an object must return a collection like a `List`, it is made immutable to the user by methods such as `Collections.immutableList()`.

5.6 Knowledge Node Network

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

The KNN layer is based around the `KnowledgeNodeNetwork` interface. Its implementation, `KnowledgeNodeNetworkImpl`, has the following fields:

`mapKN` One-to-one `Map` of input `Tags` to associated `KnowledgeNodes`.

`activeTags` `HashSet` of active `Tags`, corresponding to input `Tags` of fired `KnowledgeNodes`.

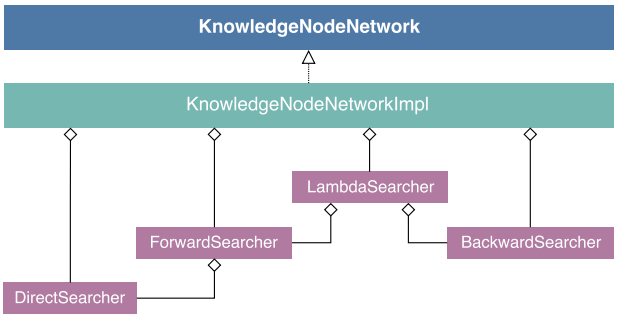


Figure 10: UML diagram of the major classes in the ***knn*** *package.*

The choice of `HashSet` for the `activeTags` field is not arbitrary. Indeed, `HashSets` were used for most of the collections with variable size in the ES and KNN. 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 `HashSets`. `HashSets` are also faster because they have $O(1)$ access time, whereas `ArrayLists` have $O(N)$, where N is the number of elements in the collection. Access time is important because the tags in the KNN and ES are accessed often. `HashSets` also have the advantage of only permitting unique elements. This is useful because there should never be two copies of the same tag or the same KN in the system.

The `KnowledgeNode` class implements the functionality of a KN, which has the following important fields to implement the functionality described in Section 2:

`inputTag` Input `Tag`.

`outputTags` `HashSet` of output `Tags`.

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

`threshold` `int` threshold that causes firing of the `KnowledgeNode`.

`strength` `int` that biases the activation of a `KnowledgeNode`, causing early firing.

`belief` `int` representing the belief that the `inputTag` is true (0 to 1).

`age` `long` representing the age of the `KnowledgeNode`.

5.6.1 Excitation

There are different ways that KN excitation could be implemented. With simple linear activation, excitation would increment the node's activation parameter, which initially starts at 0. If $activation \geq threshold$, the KN fires, causing the activation of the output tags. The activation can also be implemented using a sigmoid function, which is more representative of neurons in the brain [15]. The linear activation was chosen to be implemented for its simplicity.

5.6.2 Strength

A simple implementation of strength would be as a constant coefficient multiplying the activation parameter. So, instead of checking when the activation is greater than the threshold, one would check if $activation \times strength \geq threshold$, where the strength is positive. The strength value could be 0 however, which effectively shuts off the KN. Another way of implementing strength is to check $activation + strength \geq threshold$. In this case, strength can be negative and make the KN fire later than normal. The constant coefficient version of strength was implemented.

5.6.3 Search

The most important methods in the `KnowledgeNodeNetwork` are the ones relating to searching and thinking, as described in Section 2. The search methods are `directSearch()`, `forwardSearch()`, `backwardSearch()` and `lambdaSearch()`. The `directSearch()` method takes as input the `Tag` to activate and returns the `Set` of `Tag`s activated as a result of exciting the `KnowledgeNode` associated with the input `Tag`. Note that the input `Tag` itself is not returned. Similarly, the other search methods take a `Set` of `Tag`s as input and return the `Set` of `Tag`s activated as a result of searching. The other search methods also take a `ply` parameter specifying the depth of the search. If `ply` is 0, the search continues until quiescence, i.e., until no `Tag`s are activated during a search cycle.

For each of the search methods, there is an associated searcher class, i.e., `DirectSearcher`, `ForwardSearcher`, `BackwardSearcher` and `LambdaSearcher`. This is a good example of the single responsibility principle. Delegating the search behavior to different classes allows for modularity and easy unit testing. Re-use of these classes was also taken advantage of, with `ForwardSearcher` using `DirectSearcher`, and `LambdaSearcher` using `ForwardSearcher` and `BackwardSearcher`, as shown in Figure 10. The `ForwardSearcher`, `BackwardSearcher` and `LambdaSearcher` also extend a `Searcher` abstract class, which handles some common functionality, i.e., searching until quiescence when the `ply` is 0.

Similarly, there are methods relating to thinking, i.e, `forwardThink()`, `backwardThink()` and `lambdaThink()`. These behave in the same way as their searching counterparts, except that they take as input all the currently active tags in the KNN, instead of user-provided input.

5.6.4 Aging

There are two main choices for the implementation of aging of the KNs:

1. Increment an age counter (from 0) for every KN every x amount of time. Once a KN is excited, its age is reset to 0. If the age exceeds a certain threshold, it is discarded. The KNs can be stored in a sorted mapping from the age counter to the KNs, where a KN is simply moved to the next slot when its age is updated.

2. The KN's age is initialized to the current UNIX time. When a KN is about to be excited, the current UNIX time is compared to the last stored one. If the difference is greater than some threshold, the KN is discarded. Otherwise, the KN's age is set to

the current UNIX time. The KNs can be stored in a sorted mapping from age to KN.

The second method from the above was chosen for the main reason that it is computationally more efficient to update a KN upon exciting it, rather than updating multiple at a time every x time interval. There is a trade-off however, since the first method has a simpler storage mechanism, dealing with either incrementing an index or resetting an index to 0 at every age update. The second method has to instead keep a sorted mapping from the UNIX time to the KNs. While this is more complex, it can be relatively simply implemented with a `TreeSet` in Java, which is what was chosen. In this way, for each search method, the youngest KNs are excited first.

To accomplish this, a new `TreeSet` field was created in the KNN named `ageSortedKNs`. This is a set of KNs sorted by their age. It is used in backward search to iterate over the KNs, and in direct search to update the age. There is also an associated age limit in the backward searcher named `backwardSearchAgeLimit` which can be set by the user. While iterating over the KNs, if any KN's age is greater than this threshold, the current ply's search terminates.

5.7 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.

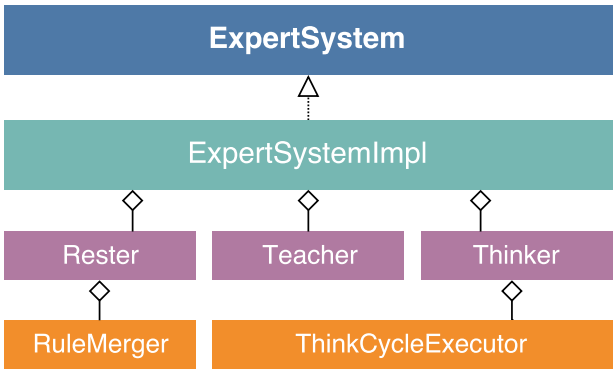


Figure 11: UML diagram of the major classes in the `es` package.

The ES layer is based around the `ExpertSystem` interface. Its implementation, `ExpertSystemImpl`, has the following important fields:

`readyRules`	`HashSet` of `Rules` that have not been activated yet.
`activeRules`	`HashSet` of active `Rules`.
`facts`	`HashSet` of active `Facts`.
`recommendations`	`HashSet` of active `Recommendations`.

The most important method in the `ExpertSystem` is `think()`, which implements the functionality described in Section 2. It has the following parameters:

`ply`	The amount of thinking cycles to execute.
`generateRule`	If `true`, generates a `Rule` based on the active `Facts` and the ones activated as a result of thinking.

The `think()` method has an associated `Thinker` class, whose sole purpose is to execute thinking cycles by using the `ThinkCycleExecutor` class. It does so by iterating over the ready rules in the ES and and activating those whose input facts are active.

The `rest()` method allows the creation of new rules based on existing ones, with the associated `Rester` class, which in turn uses the `RuleMerger` classes. It merges rules when one rule implies another. For example, if we have $P(A) \rightarrow P(B)$ and $P(B) \rightarrow P(C)$, then a new rule can be created: $P(A) \rightarrow P(C)$.

The `teach()` method parses simple *if-then* sentence `Strings` to generate rules by using the `Teacher` class. For example, the sentence *if* $P(A)$ *then* $P(B)$ would be converted to a rule of the form $P(A) \rightarrow P(B)$. The currently supported words denoting rule input are *if*, *when*, *while* and *first*. The words denoting rule output are *then*, *next* and *do*.

5.8 Visualization

The Java graph visualization library GraphStream was used to visualize the KNN and the process of searching or thinking through it. The graphing code created was placed in the `graphing` package. A UML diagram of the important graphing dependencies can be seen in Figure 12. A legend for reading the produced graph can be seen in Figure 13. The small nodes represent tags and the bigger nodes represent KNs. The red nodes indicate either active tags or fired KNs. Non-fired KNs are blue. Facts are purple, rules are teal and recommendations are orange. The visual tests for this graphing tool were placed in the test directory and will be discussed in Section 6.

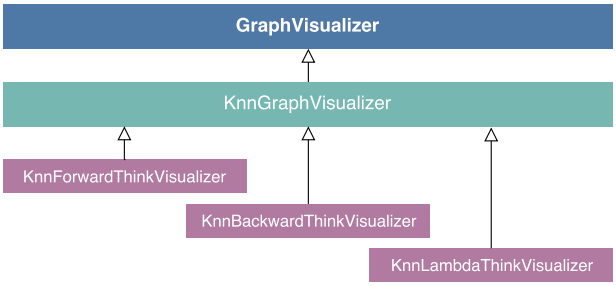


Figure 12: UML diagram of the important classes in the ***`graphing`*** *package.*

5.9 Documentation

Extensive documentation was created as comments in the code and converted to Javadoc. This can be found in the `docs` directory of the project, or hosted online[3]. Instructions are also provided in the README located in the top-level directory of the project.

6 Results and Tests

Unit and integration tests in Prometheus were conducted using the TestNG framework in Java, which provides a simple and intuitive way to create assertions in tests. These are run in the continuous integration suite provided by Travis CI on the GitHub repository. This enforces a level of quality for the code when someone pushes a change. Some visual tests using the KNN graphing tool were also created. All tests were placed in the test directory of the project.

6.1 Unit Tests

Unit tests for individual methods and classes were constructed using TestNG and Mockito. Mockito allows easy mocking of dependencies. These tests were created in parallel with writing the functional code, to ensure proper functionality.

The unit tests are behavior-driven and written in such a way that someone reading them can easily understand the intended behavior of the element under test. The test methods follow the `mustAction()` naming convention. For example, the `RuleMergerTest` class has a `mustMakeEmptyRule()` method, which tests that the merger returns an empty `Optional` object when there are no rules to merge.

The body of each unit test method has three sections: *given*, *when* and *then*. The *given* section sets up the intended return values of the dependencies mocked by Mockito, if present. The *when* section executes the actual method under test. The *then* section contains the TestNG assertions concerning the result of executing the method under test.

The package location of each unit test mirrors the location of the class under test in the main code. For example, the `BackwardSearcher.java` file is located in `src/main/java/knn/internal/` and its unit test, `BackwardSearcherTest.java`, is therefore located in `src/test/java/knn/internal/`. This allows the unit tests to access package-private classes and methods.

6.2 Integration Tests

Integration tests were also created to test end-to-end behavior. These are more high-level than unit tests and simply test input and output of methods without mocking their dependencies. All integration tests are located in

[3]The Javadoc can be found here:
`http://seanstappas.me/prometheus-ai/`

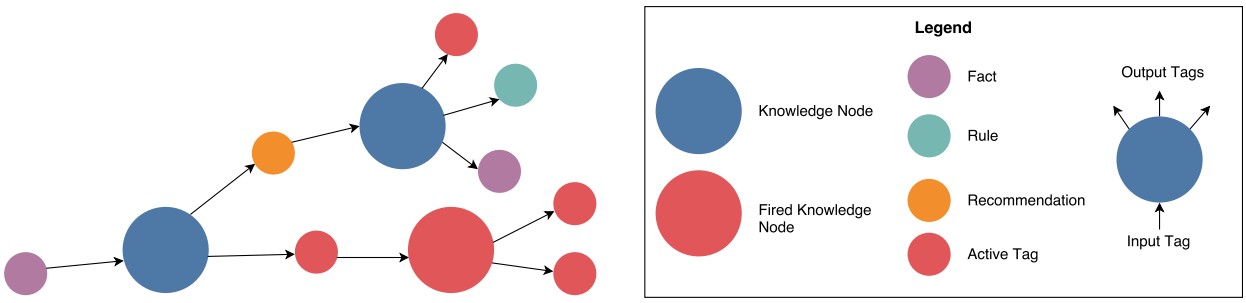


Figure 13: Legend for reading the graph produced by `KnnGraphVisualizer`.

Table 3: Test setup for `testES()`. *Middle activation steps omitted.*

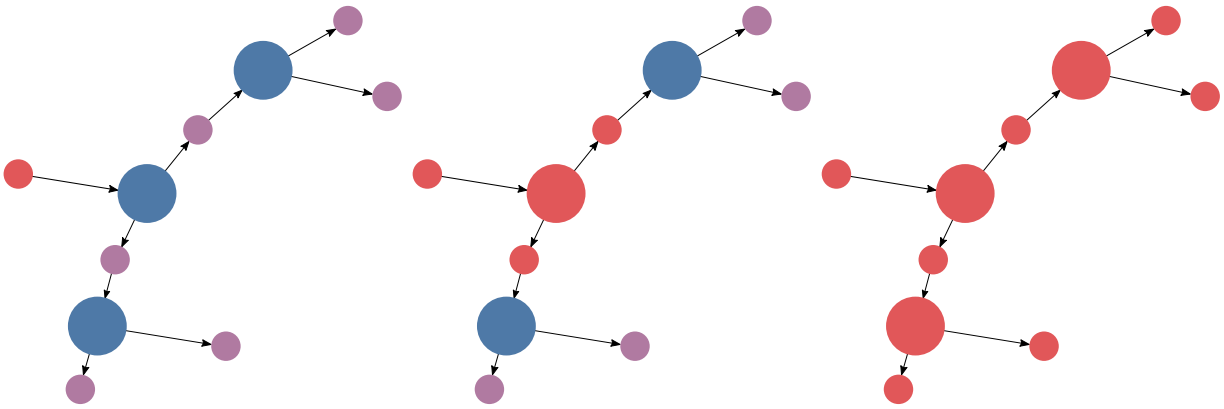
State	Ready Rules	Active Rules	Active Facts	Active Recommendations
Initial	$A, B \to D$ $D, B \to E$ $D, E \to F$ $G, A \to H$ $E, F \to @Z$		A, B	$@X, @Y$
⋮	⋮	⋮	⋮	⋮
Final	$G, A \to H$	$A, B \to D$ $D, B \to E$ $D, E \to F$ $E, F \to @Z$	A, B $D, E,$ F	$@X, @Y, @Z$

`src/test/java/integration/`. Tests for the ES, KNN and both combined can be found here.

For example, the test setup for an ES integration test can be seen in Table 3. The columns from left to right (ignoring *State*) correspond to the elements in `readyRules`, `activeRules`, `facts` and `recommendations`, respectively. The first row corresponds to the initial state of the ES and the last row corresponds to the expected final state. The presence of both the initial and final states are asserted with TestNG. This test can be found in the `SimpleExpertSystemTest` class.

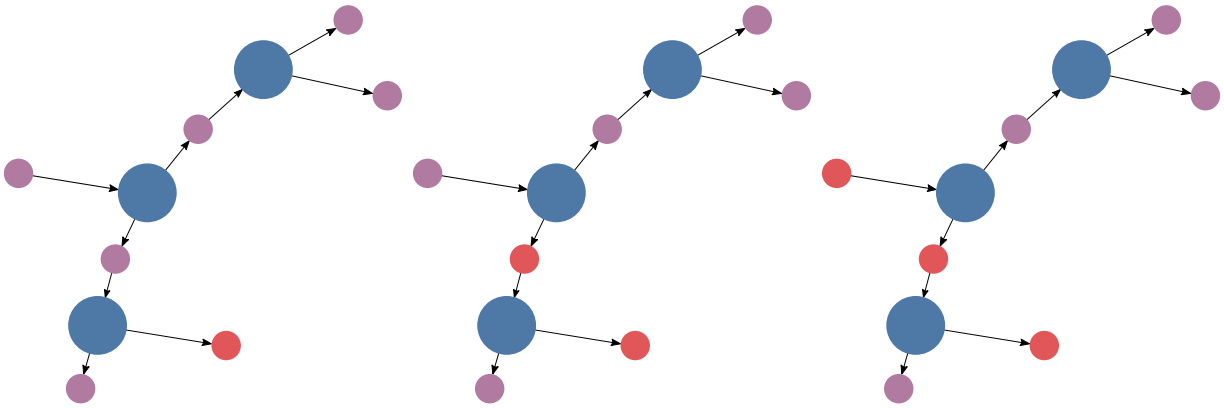
6.3 Visual Tests

Some visual tests were carried out with the created KNN visualization tool. These can be found in `src/test/java/graphing/`. Visualizations of forward, backward and lambda search can be seen in Figures 14 to 16. These tests can be found in `KnnSimpleForwardThinkVisualizer.java`, `KnnSimpleBackwardThinkVisualizer.java` and `KnnSimpleLambdaThinkVisualizer.java`. Refer to Figure 13 for the meaning of the node colors. In these tests, a threshold of 1 was used for simplicity and the initial active tags are provided as search input.



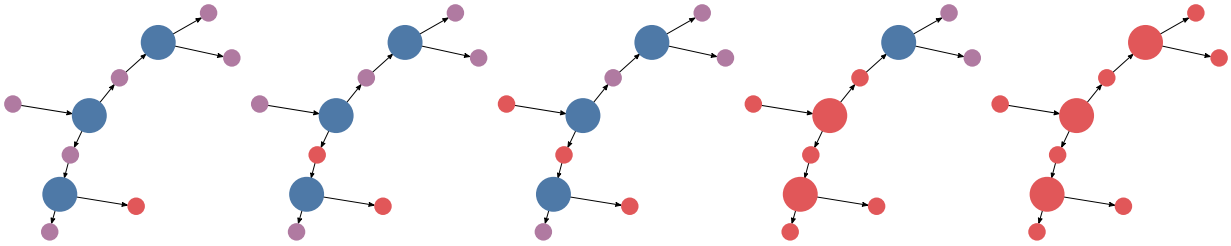
(a) Iteration 0. *(b) Iteration 1.* *(c) Iteration 2.*

Figure 14: Forward search visualization in the KNN.



(a) Iteration 0. *(b) Iteration 1.* *(c) Iteration 2.*

Figure 15: Backward search visualization in the KNN.



(a) Iteration 0. *(b) Iteration 1.* *(c) Iteration 2.* *(d) Iteration 3.* *(e) Iteration 4.*

Figure 16: Lambda search visualization in the KNN.

7 Impact on Society and the Environment

7.1 Use of Non-renewable Resources

As purely a software project, there are no physical materials needed to construct this system. For this reason, less emphasis will be put on this section. The only related physical resources potentially needed are the materials needed to construct the robots and the computers to house the software. One area of concern could be the energy source of the robots themselves, which

would probably vary depending on context. Ideally, the source should be a renewable one. For example, solar panels could be placed on the robots to provide energy.

7.2 Environmental Benefits

Prometheus could be used in a context beneficial to the environment. For example, the system could be used as a tool to control robots after oil spills, where the robots could theoretically safely contain the problem faster than humans and thus limit the risk on the environment. There is already research being done on employing robots in this context. Indeed, MIT's Senseable City Lab has been working on Seaswarm, a system composed of a fleet of vehicles to help clean future oil spills [16]. An AI like Prometheus to control a swarm of robots like this could be very valuable.

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. Indeed, the US Department of Defense has plans to employ AI for autonomous weapons to attack targets without human intervention in the future [17]. This would have the advantage of potentially saving human soldiers' lives [18]. However, there are also concerns because this would make it much easier to start a battle. More than 3000 AI and robotics researchers have signed an open letter arguing against a military AI arms race [19]. They argue that autonomous weapons would not be beneficial to society, since they would be ideal for assassinations, destabilizing countries or selectively subduing or killing a population.

Another possible issue is the future loss of jobs to be done by humans, with automated systems like Prometheus replacing physical labour. This problem can be solved as a society, perhaps by having more support for universal basic income (UBI). Telsa CEO Elon Musk is a proponent of the idea, saying that UBI will be necessary in the future [20]. Microsoft co-founder Bill Gates suggests possibly having a tax on robots to help pay for this income [21].

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 or animals are? The issue has been mentioned by the Institute for the Future, calling for the possibility of "robo-rights" in the future [22].

7.4 Benefits to Society

This type of system could be extremely useful in many contexts in society. For instance, the system could be used to send and control robots in an area that would otherwise be very dangerous for humans. For example, robots are often used in the aftermath of nuclear disasters to prevent the unnecessary loss of human life. Indeed, in the Chernobyl nuclear disaster, the Soviet authorities employed the use of robots to avoid losses of human life [23]. With an AI like Prometheus, the robots could be controlled in an intelligent manner.

The system could also be used to further space exploration, with an AI controlling multiple robots exploring the surface of Mars, for instance. The value of AI in this context has been clearly shown, with the Mars Curiosity rover recently being upgraded to have its own AI system using computer vision to identify rocks [24]. This system can therefore help further important cutting-edge research in space exploration.

8 Conclusion

The Expert System (ES) and Knowledge Node Network (KNN) layers of the Prometheus AI model were completed in Java based on design criteria and feedback from the supervisor. These layers were tested using the TestNG Java framework with positive results and extensive Javadoc

was produced. Possible impacts on the environment and society were also discussed.

Possible future work would include finalizing the entire system by building the Neural Network (NN) and Meta Reasoner (META) layers. It would also include implementing more complex features, such as attention and learning in the KNN.

References

[1] J. Vybihal, "Full AI model," 2016.

[2] S. Stappas, "Prometheus AI: Phase 1," 2017.

[3] J. Vybihal, "Knowledge nodes," 2017.

[4] D. O. Hebb, *The organization of behavior: A neuropsychological theory.* Psychology Press, 2005.

[5] T. Mitchell, *Machine Learning*, ser. McGraw-Hill international editions - computer science series. McGraw-Hill Education, 1997. [Online]. Available: https://books.google.ca/books?id=xOGAngEACAAJ

[6] H. L. Roediger and K. B. McDermott, "Creating false memories: Remembering words not presented in lists," *Journal of experimental psychology: Learning, Memory, and Cognition*, vol. 21, no. 4, p. 803, 1995.

[7] S. E. Taylor, *Positive illusions: Creative self-deception and the healthy mind.* Basic Books, 1989.

[8] D. C. Knill and A. Pouget, "The Bayesian brain: the role of uncertainty in neural coding and computation," *Trends in neurosciences*, vol. 27, no. 12, pp. 712–719, 2004.

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

[10] D. Kahneman, *Thinking, fast and slow.* Macmillan, 2011.

[11] M. P. Mattson and T. Magnus, "Ageing and neuronal vulnerability," *Nature Reviews Neuroscience*, vol. 7, no. 4, pp. 278–294, 2006.

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

[13] J. L. Popyack, "Object Oriented Programming," https://www.cs.drexel.edu/~introcs/Fa15/notes/06.1_OOP/Advantages.html?CurrentSlide=3, 2015.

[14] R. C. Martin, "Getting a solid start. - clean coder," Dec 2009. [Online]. Available: https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start

[15] J. Palka and C. Stevens, "Neurophysiology — a primer," 1967.

[16] "A swarm of robots to clean up oil spills," https://sap.mit.edu/article/standard/swarm-robots-clean-oil-spills, December 2010.

[17] D. Gershgorn, "The US army wants to weaponize artificial intelligence — Quartz," https://qz.com/767648/weaponized-artificial-intelligence-us-military/, August 2016.

[18] B. Allenby, "The difficulty of defining military artificial intelligence," http://www.slate.com/articles/technology/future_tense/2016/12/the_difficulty_of_defining_military_artificial_intelligence.html, December 2016.

[19] "Open letter on autonomous weapons - future of life institute," https://futureoflife.org/open-letter-autonomous-weapons/.

[20] A. Kharpal, "Tech CEOs back basic income as AI job losses threaten industry backlash," http://www.cnbc.com/2017/02/21/technology-ceos-back-basic-income-as-ai-job-losses-threaten-industry-backlash.html, February 2017.

[21] "Bill Gates: Job-stealing robots should pay income taxes," http://www.cnbc.com/2017/02/17/bill-gates-job-stealing-robots-should-pay-income-taxes.html, February 2017.

[22] "Robots could demand legal rights," http://news.bbc.co.uk/2/hi/technology/6200005.stm, December 2006.

[23] G. Dvorsky, "A museum of robotic equipment used during the Chernobyl disaster," http://io9.gizmodo.com/a-museum-of-robotic-equipment-used-during-the-chernobyl-512831778, 2013.

[24] "How does mars rover Curiosity's new AI system work? — astronomy.com," http://www.astronomy.com/news/2016/08/how-does-mars-rover-curiositys-new-ai-system-work, August 2016.

A Diagrams

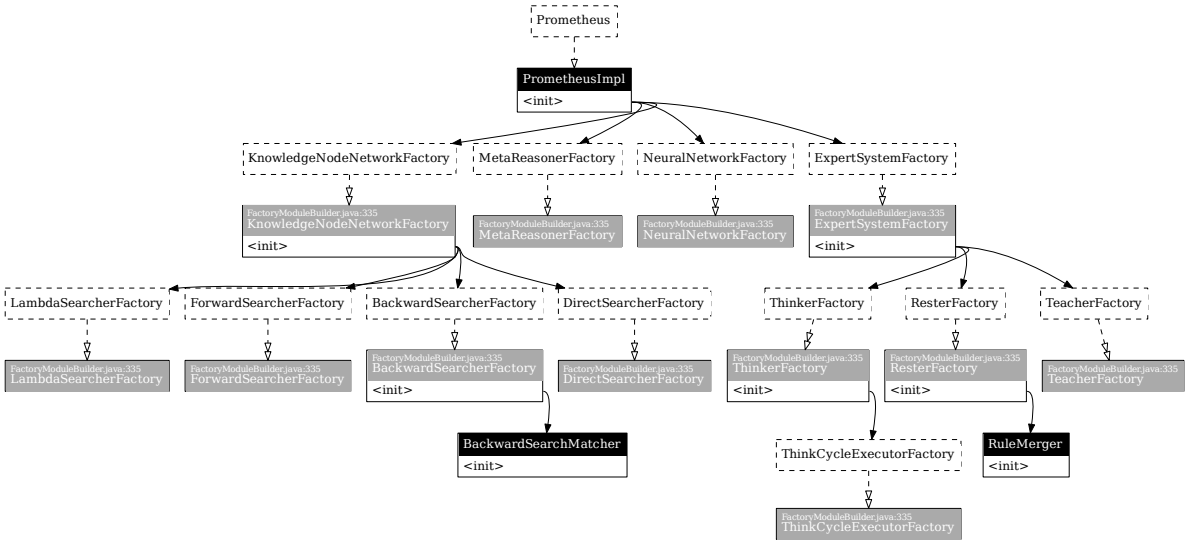


Figure A.1: Guice dependency graph.