Sean Stappas — Supervised by Prof. Vybihal

Prometheus AI

McGill

Introduction & Background

Prometheus is an AI system designed to control and coordinate multiple **robots** to achieve some goal. Its architecture is loosely inspired from the structure of the human brain, and is composed of the following four layers: the **Neural Network (NN)**, the **Knowledge Node Network (KNN)**, the **Expert System (ES)**, and the **Meta Reasoner (META)**. The **Neural Network** layer is the interface to the robots' sensors and provides informational tags to the KNN.

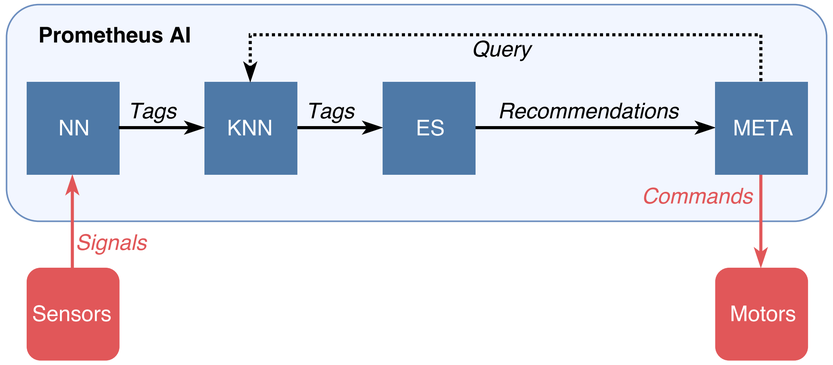
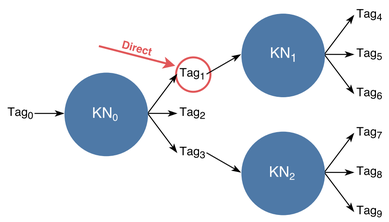
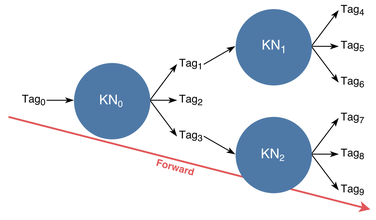


Figure 1: Prometheus AI model with labeled input and output.

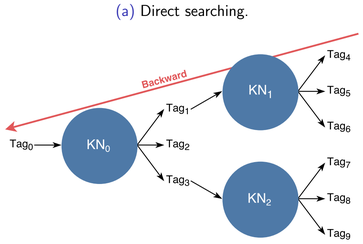
The **Knowledge Node Network** layer is the analog to **memory** in the human brain. It takes in the tags provided by the NN and outputs tags based on its knowledge. It consists of interconnected **Knowledge Nodes** (KNs), which are abstract structures representing **memories** and their connections to other memories. There are many concepts associated with KNs, such as **activation**, **firing**, **strength**, **aging** and **belief**. The KNN has 4 ways of searching to fire KNs and activate tags: **direct**, **forward**, **backward** and **lambda**.



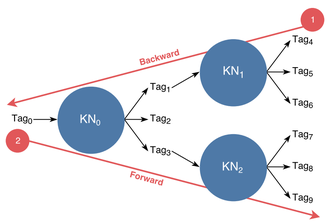
(a) Direct searching.



(b) Forward searching/thinking.



(c) Backward searching/thinking.



(d) Lambda searching/thinking.

Figure 2: Methods of searching and thinking in the KNN.

The **Expert System** 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**. Based on these rules, it performs a thinking routine to activate recommendations and provide them to the Meta Reasoner.

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.

(a) Examples of simple facts in the ES.

$$Fact := P(A_1, \ldots, A_n)$$

(b) The form of a fact in the ES. $\boldsymbol{P}$ is a predicate, and $\boldsymbol{A_i}$ is an argument.

$$Recommendation := @P(A_1, \ldots, A_n)$$

(c) The form of a recommendation in the ES. $\boldsymbol{P}$ is a predicate, and $\boldsymbol{A_i}$ is an argument..

$$Rule := Fact_1 \ldots Fact_m \rightarrow Tag_1 \ldots Tag_n$$

(d) The form of a rule in the ES, with $\boldsymbol{m}$ input facts and $\boldsymbol{n}$ output tags. The output tags must either be facts or recommendations.

Figure 3: Facts, recommendations and rules in the ES.

The **Meta Reasoner** represents high-level reasoning in the human brain. It is aware of its environment and context, and makes informed decisions to send actuator commands to the robots.

Problem

There are two main assigned tasks over the past two semesters:

1. **Design** and implement the **Expert System** and **Knowledge Node Network** in Java.
2. **Supervise** undergraduate students working on Prometheus.

Design & Implementation

The high-level **Prometheus** interface has dependencies on the `NeuralNetwork`, `KnowledgeNodeNetwork`, `ExpertSystem` and `MetaReasoner` interfaces. The `KnowledgeNodeNetwork` and `ExpertSystem` layers are the ones that are currently implemented. The code base was designed to be **object-oriented**, **efficient**, **modular** and easily **testable**. The **tags** used throughout the system are implemented using a `Tag` abstract class, with `Predicate` and `Rule` subclasses. The `Predicate` class is an abstract superclass of the `Fact` and `Recommendation` classes and represents the possible output tags of a rule.

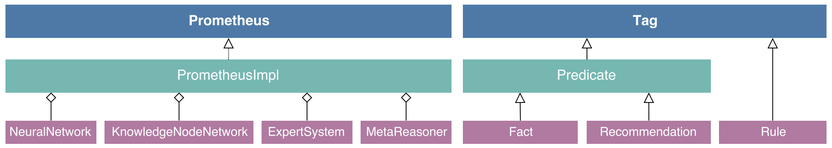


Figure 4: UML diagrams of the Prometheus and Tag packages.

The most important method in the **Expert System** is `think()`, which executes **thinking cycles** and outputs recommendations. This is performed by the `Thinker` class and its associated `ThinkCycleExecutor`. The `rest()` method allows the creation of **new rules** with the `Rester` and `RuleMerger` classes. The `teach()` method parses simple "if-then" **sentences** to generate rules by using the `Teacher` class.

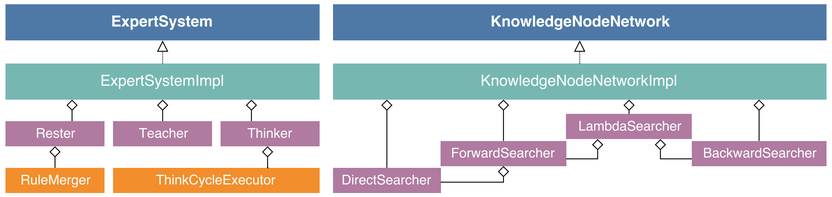


Figure 5: UML diagrams of the Expert System and KnowledgeNodeNetwork packages.

Searching in the **Knowledge Node Network** is done by the `directSearch()`, `forwardSearch()`, `backwardSearch()` and `lambdaSearch()` methods. These are controlled by the `DirectSearcher`, `ForwardSearcher`, `BackwardSearcher` and `LambdaSearcher` classes, respectively. These **modular** classes allowed great **re-use**, as can be seen in Figure 6. An interface to **load networks** from file was also created. KNs are implemented as `KnowledgeNode` objects. The **age** of a KN is implemented with UNIX timestamps, with old KNs being deleted. There is also a **belief** value associated with every KN which originates from the confidence assigned by the NN.

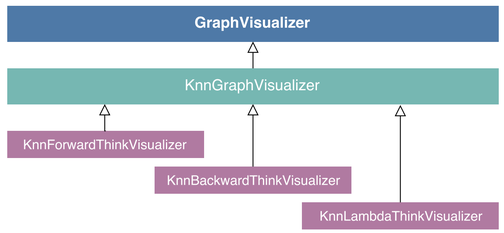


Figure 6: UML diagram of the graphing package.

The Java **graph visualization** library GraphStream was used to visualize the KNN and the process of searching through it. At each thinking or searching cycle, the graph can be updated in the UI and saved to SVG. Separate visualizers were created for **forward**, **backward** and **lambda** searching.

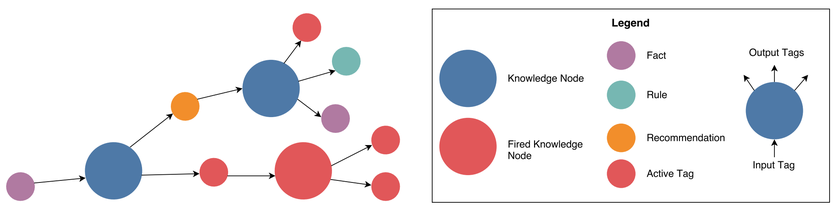


Figure 7: Legend for the graph visualization of the KNN.

Prometheus is a **Maven** project, which allows very simple use of many libraries. These libraries include: **Google Guice** for modeling and building the dependencies between the layers and their modules, **Mockito** for mocking objects in unit tests, **TestNG** for running tests and summarizing their results and **Apache Commons Lang 3** for standardized `toString()`, `hashCode()` and `equals()` methods.

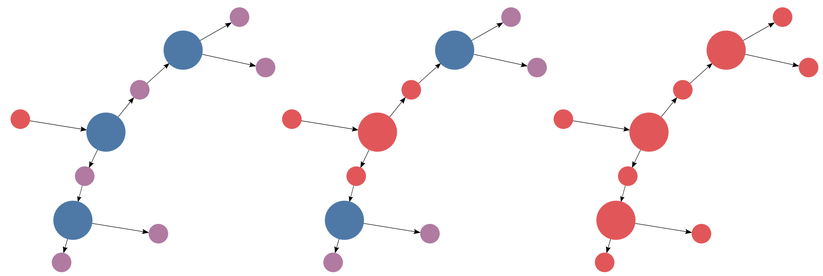
Results & Tests

Unit tests were created using **Mockito** and **TestNG**. These tests test the functionality of isolated methods. **Integration tests** were also created with **TestNG**, testing end-to-end behavior. All of these tests are part of a continuous integration test suite which runs on the GitHub repository whenever a commit is made. This is done using **Travis CI**. Figure 8 shows a simple Expert System integration test. Here, the test asserts that, given the provided initial conditions, the final state is reached correctly. Arguments are omitted for simplicity.

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

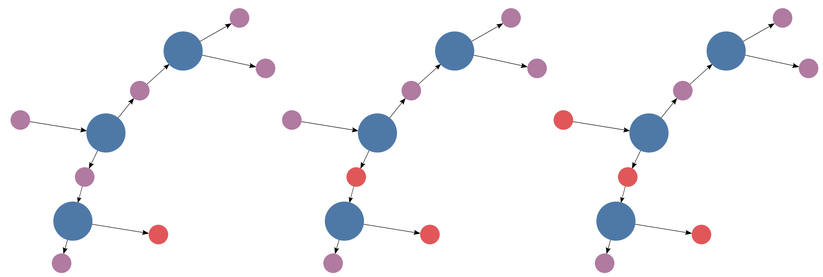
Figure 8: ES test setup representing simple rules and facts that must be brought to quiescence.

The `forwardSearch()`, `backwardSearch()` and `lambdaSearch()` were visualized using the graph visualization tool. These can be seen in Figures 9 to 11. Please refer to Figure 7 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.



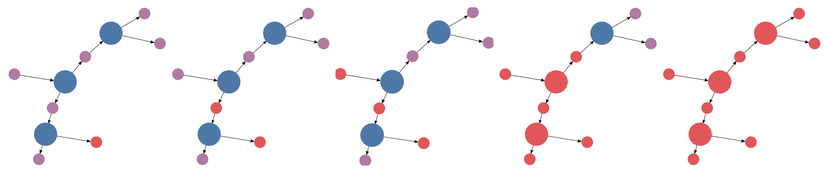
(a) Ply 0. (b) Ply 1. (c) Ply 2.

Figure 9: Forward search visualization in the KNN.



(a) Ply 0. (b) Ply 1. (c) Ply 2.

Figure 10: Backward search visualization in the KNN.



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

Figure 11: Lambda search visualization in the KNN.

Conclusion

Over the course of this project, I learned many skills relating to **designing** and **implementing** large-scale software projects. I also learned a great deal about various **cognitive science** topics. Setting up systems for software project **collaboration** was also learned. Possible further work relating to this project could be implementing the **NN** and **META** layers, as well as more features in the KNN, such as **learning** and **attention**.