Gauntlet Memory Subsystem

Self-Organizing Maps and More

Sam Glidden (sglidden@mit.edu)
28 April 2008
Supervisor: Professor Patrick Winston

Files

Key files within the Memory system include:

1. **Memory.**java – "top-level" of the memory; provides a single interface through which the rest of Gauntlet can access the self-organizing maps (SOMs) and other features.
2. **SomSpace.**java – a container from the various SOMs. Each representation type ("frame") should have a SOM in SomSpace.
3. **ThingSpace**.java -- stores ever Thing that memory has seen. It also stores relations between Things, such as a casual relation provided by a "Time" Frame.
4. **Som**.java – an interface (in the Java sense) that must be implemented by any SOM that SomSpace will use.
5. **NewSom**.java – an implementation of Som.java that most representation types currently use
6. **mergers/** -- a folder that contains static methods that can be used by specific implementations of SOMs to "merge" or combine Things.
7. **metrics/** -- a folder that contains static methods that can be used by specific implementations of SOMs to provide distance metrics between Things.
8. **utilities/** -- a folder that contains static methods for distance metrics and Thing manipulations.

Interfacing with the Memory System

There are two ways to interface with the Memory subsystem. Each way requires dependency only on Memory.java.

Way 1: Wires

Everything in Gauntlet outside of the Memory can use the "Wire" mechanism (http://groups.csail.mit.edu/genesis/wire.html) to connect to Memory.java. From there, you can request information from the self-organizing maps.

The memory has different ports, each for a different purpose.

```
PORT_REPTREE = "input a rep tree from an entire sentence";
PORT_STIMULUS = "Input stimulus port";
PORT_RESPONSE = "Response to stimulus output port"
PORT_PREDICTIONS = "output port for predictions";
```

To provide the Memory system with a representation Thing to be stored in a SOM, use *PORT_REPTREE*. To get the neighbors (i.e. similar elements, as determined by the appropriate SOMs) of a Thing, use *PORT_STIMULUS*. The answer will be sent out on *PORT_RESPONSE.*

An example of this is found within Guantlet.java:

```
// connect input to memory
Connections.wire(getMemoryGate(), Memory.PORT_REPTREE, getMemory());

// connect question expert to memory system
Connections.wire(QuestionExpert.MEMORY_PORT, getQuestionExpert(),
      Memory.PORT_STIMULUS, getMemory());
Connections.wire(Memory.PORT_RESPONSE, getMemory(),
      QuestionExpert.MEMORY_PORT, getQuestionExpert());
```

Furthermore, you can request an entire SOM (for instance, for GUI display purposes), like this:

```
Connections.wire(TrajectoryFrame.FRAMETYPE, getMemory(),
      getSomTrajectoryViewer());
```

Finally, *PORT_PREDICTIONS* is an output port. Whenever the Memory receives an input that stimulates it to make a prediction (an expectation of seeing a Thing in the future), it will output that expected Thing on this port.

Further ports can be added easily: just stick them in Memory.java. Memory.java can then call methods in SomSpace.java or elsewhere as appropriate. Keep in mind that the memory system is multithreaded—you will probably want to spawn a new Thread to handle the request.

Way 2: Method Calls

If the overhead of using wires is not appropriate for your application, you can simply call a number of useful methods directly. From wherever you need to use the Memory, use the static singleton constructor:

```
Memory mem = Memory.getMemory();
```

Then, you can call any public method by doing `mem.methodName(params)`. Particularly useful public methods are:

```
public void clearSOMs()
public void addRepTree(final Thing t)
public List<Thing> getNeighbors(final Thing t)
public List<Thing> getNearest(final Thing t, int n)
public boolean containsInSOMs(Thing t)
public int getFrequencyInSOMs(Thing t)
```

These are all well-documented in the code.

Adding New Representations

It is fairly simple to add another representation to the memory system. The basic steps are:

1. Create a SOM to hold the new representation in SomSpace.java. For example:

```
soms.put(TrajectoryFrame.FRAMETYPE, new NewSom<Thing>(new
      ThingMetric(), new ConditionalMerger(), .3));
```

2. Make sure the representation type is registered in RecognizedRepresentations.java within the ALL_THING_REPS List.

Debugging

Memory.java has a static Boolean flag called “DEBUG”. If you set this to true, the memory system will print out useful status messages. If you add further println()’s to any files, please wrap them in an if that checks Memory.DEBUG.

Description of NewSom.java

While any class that implements Som.java can be used in SomSpace, in actuality most SOMs are of a type called NewSom. A NewSom requires a Metric and a Merger, found in the respective folders, to describe how the SOM’s contents are compared and merged. Essentially, the Metric is used to determine the distance between elements in NewSom, while the Merger is used to modify current elements when a new one is added.

The standard ThingMetric.java and ConditionalMerger.java, used by most instances of NewSom, are flexible enough to process any Thing. Thus, you do not need to create a new Merger or Metric when you add a new representation type to the system—unless, that is, you need special, idiosyncratic behavior.

NewSom also has a concept of quantity. The same element can be found within NewSom multiple times. When that element is slated for modification (i.e. a merge caused by the addition of a new element), only one of that element is actually changed. The remainders are left untouched. The motivation behind this behavior is the idea that frequent experiences should not be quickly forgotten.