Aula 13 – LIDA 2

Atividade 1

Foi feito o download e extração da versão 1.2 do framework LIDA que será usado para a execução dos tutoriais fornecidos pelo professor.

Atividade 2 – Tutorial Project 1

1) Basic Agent Exercise 0

Introdução

Este exercício tem como objetivo uma exploração superficial da estrutura do projeto deste tutorial.

a) Arquivos de código explorados

- **myagent.Run**
  É a onde o método *main* do projeto se encontra. É lá que a classe *AgentStarter* explicada no tutorial do relatório anterior é invocado, dando início à execução do agente.

- **myagent.modules.ButtonEnvironment**
  Contém a classe *ButtonEnvironment*, que implementa a interface *EnvironmentImpl.* Como citado no relatório anterior, implementar esta classe é necessário para que se possibilite a percepção do ambiente pelo agente e a ação deste sobre o primeiro. Podemos observar que, entre outros, os métodos *getState* e *processAction* são sobrescritos de forma a possibilitar essas ações.

  No escopo deste agente, *getState* retorna um recorte de imagem do ambiente onde as formas são exibidas. O agente passa o retângulo que deseja 'enxergar' e o ambiente retorna este recorte.

Depois de processar esta informação, o agente utiliza *processAction* para executar uma ação. Neste escopo, as ações válidas são pressionar um dos dois botões ou soltar todos eles. Dependendo da ação enviada, o ambiente atualiza uma propriedade chamada *lastPressedButton*, sendo portanto alterado pelo agente.

- **myagent.modules.ButtonSensoryMemory**
  Contém uma classe *ButtonSensoryMemory*, que é a implementação da interface SensoryMemoryImpl (a memória sensorial citada no relatório anterior) para este escopo de aplicação.
  Ela tem um método *runSensors* que executa os sensores do agente, o que consiste basicamente em acessar o ambiente através do método *getState* citado acima e armazenar a imagem localmente.
  Com uma imagem armazenada desta forma, outros módulos podem, através do método *getSensoryContent*, recuperar um pixel de cor desta imagem ou seu conteúdo na íntegra, dependendo modo (*color* ou *all)* passado como parâmetro.

- **myagent.featuredetectors.ColorDetector**
  Contém a classe ColorFeatureDetector que estende a classe *BasicDetectionAlgorithm.* Apenas os métodos *init* e *detect* são sobrescritos. O primeiro configura seu modo de detecção para *color* e determina a posição (50,50) como alvo. O segundo acessa a memória sensorial através de *getSensoryContent* utilizando esta configuração para recuperar a cor do pixel da imagem armazenada na memória sensorial nesta posição.

- **myagent.featuredetectors.ShapeDetector**
  Contém a classe ShapeFeatureDetector que estende a classe *BasicDetectionAlgorithm.* Apenas os métodos *init* e *detect* são sobrescritos. O primeiro configura seu modo de detecção para *all*, e a área do botão buscado para 1000 (pixels), além da cor de fundo do ambiente. O segundo acessa a memória sensorial através de *getSensoryContent* utilizando esta configuração para recuperar a imagem do ambiente como um todo e faz uma conta simples que computa a quantidade de pixels de cor diferente da do fundo para determinar se a forma é um quadrado ou um círculo.

b) Arquivos de configuração
   - lidaConfig.properties

Contém os caminhos para os arquivos de configuração basicAgent.xml, factoryData.xml, guiPanels.properties, guiComands.properties e logging.properties, além da propriedade gui.enabled configurada como *true* de forma que a GUI será exibida

- basicAgent.xml
  Possui as configurações para o *taskManager*, *taskSpawner*, *submodules* e *listeners*, como as classes que devem ser usadas em cada um dos elementos e sub-módulos e configurações específicas do ambiente e de comportamento de cada um deles.

- factoryData.xml
  Contém as definições dos elementos que devem ser carregados e disponibilizados no momento da execução. Contém as configurações para *strategies, nodes, links* e *tasks*

- guiPanels.properties
  Contém as configurações da interface gráfica utilizada durante a simulação. Entre os parâmetros se encontra a definição do ambiente para utilizar o ambiente do tutorial (*myagent.ButtonEnvironment* e interface associada *myagent.guipanels.ButtonEnvironmentPannel*)

2) Basic Agent Exercise 1

Introdução

Este exercício é um guia básico da GUI da arquitetura apresentada quando o tutorial é executado.

a) Gerenciamento da execução

A GUI da arquitetura permite que o usuário controle o fluxo de execução de seu agente.
É possível dar continuidade ou pausar a execução utilizando o botão *start/pause*. Enquanto estiver em estado de execução, o valor do *tick* do *TaskManager* é exibido em uma caixa de texto (*current tick)*.

Se for desejado que apenas um determinado número de ciclos sejam executados a partir do estado atual, pode-se ativar o botão *step mode*, que executa exatamente a quantidade de ciclos (ou *ticks)* determinado na caixa de texto ao seu lado. O botão *Run Ticks* inicia a execução até que o número de *ticks* seja executado.
Além disso, pode-se facilmente alterar a duração de cada *tick* em tempo de execução através da caixa *Tick duration*, que determina a quantidade de milissegundos que um *tick* deve levar no mínimo (devemos levar em consideração que dependendo da complexidade das operações de um agente é possível que um ciclo dure mais que o tempo mínimo de um *tick*).

b) Gerenciamento e visualização dos Logs

A aba *logging* mostra os logs gerados pelo agente em tempo de execução. É possível filtrar os logs pela classe de origem e tipo de log utilizando os *drop downs* Logger e Logging Level.

c) Arquivo de configuração primário

Através da aba *Configuration Files*, podemos visualizar e editar os parâmetros existentes no arquivo de configuração primária da aplicação. Como citado no relatório da aula passada, este arquivo contém basicamente o caminho para os arquivos de configuração secundários e uma propriedade que define se a interface deve ou não ser executada quando o agente é inicializado.

3) Basic Agent Exercise 2

Introdução

Neste exercício é demonstrado como se faz possível a visualização do estado interno do agente utilizando a interface gráfica fornecida pelo Framework e como interpretar a informação. Antes de se inicializar o agente, trocamos o arquivo de configuração do agente para um que não possui configurações para detecção da forma círculo nem da cor azul. Desta forma, o agente percebe apenas quadrados vermelhos.

- *PAM table GuiPannel'*

Exibe o valor de ativação atual, base e limite de cada nó da memória associativa perceptual. Durante a execução, podemos ver que os valores dos nós *square* e *red* são modificados quando um quadrado vermelho aparece no ambiente. Os valores *blue* e *circle* não são modificados justamente porque as tarefas responsáveis por sua percepção foram removidas da configuração do agente.

- *PerceptualBuffer GuiPannel*
  Mostra os nós do que possuem ativação atual maior que zero. Se colocarmos o mouse sobre um nó podemos ver o valor de ativação atual do mesmo. Supostamente, estes nós representam o *PerceptualBuffer* do *Workspace*, uma vez que continuam existindo temporariamente mesmo quando o nó percebido não está mais no ambiente mas que decaem com o tempo e são removidos uma vez que sua ativação se anula.

- *GlobalWorkspace GuiPannel*
  Nesta aba, são exibidas as coalisões existentes no momento no *Global Workspace* assim como seus valores de ativação e um histórico das últimas coalisões que 'venceram' a competição e foram difundidas através do sistema. Estas coalisões são adicionadas pelo *Workspace* no *GlobalWorkspace*, de onde, segundo o modelo LIDA, são difundidas para todo o sistema. Contudo, na implementação de um agente utilizando o Framework, a informação será enviada para aqueles módulos que registraram *listeners* para o *GlobalWorkspace*.

- *CurrentSituationalModel GuiPanenel*
  O tutorial nos auxilia e guia na inserção de uma aba para o módulo *CurrentSituationalModel* através da alteração da configuração do arquivo de configuração *guiPanels.properties*. Nesta nova aba podemos ver a representação do que o agente está compreendendo da situação atual do ambiente. Esta informação se forma da associação dos elementos no *Perceptual Buffer* do *Workspace* com aqueles na Memória Episódica Transiente e Memória Declarativa. O conteúdo do *Perceptual Buffer*, por sua vez, é proveniente da Memória Perceptual Associativa.

4) Basic Agent Exercise 3

Introdução

Neste exercício é demonstrado os passos para configuração de um agente desde a definição de um ambiente com o qual ele interage até a formação de coalisões em seu Workspace Global que refletem os elementos percebidos e identificados no ambiente.

Definição do ambiente

Consiste na definição do módulo que implementa o ambiente no qual o agente está inserido. No tutorial, já é fornecida a implementação da simulação do ambiente. A tarefa proposta é definir este ambiente como aquele utilizado na simulação. Para isso, na seção de sub-módulos do arquivo de configuração do agente, definimos a tag *Environment* determinando a classe que irá ser utilizada e os parâmetros que ela utilizará.

```
<submodules>
        <module name="Environment">
                <class>myagent.modules.ButtonEnvironment</class>
                <param name="height" type="int"> 10</param>
                <param name="width" type="int"> 10</param>
                <taskspawner>defaultTS</taskspawner>
        </module>
                ...
</submodules>
```

Para que isso funcione, é necessário que a classe referenciada implemente a interface EnvironmentImpl, como é o caso de ButtonEnvironment. Esta classe por sua vez, determina aleatoriamente o que fazer a cada momento, podendo limpar o ambiente e não desenhar nenhum elemento ou limpar e desenhar um círculo azul ou um quadrado vermelho. Neste ponto do exercício não existe nenhum link entre a Memória Perceptual Associativa e o Workspace de forma que quaisquer elementos que possam ser identificados pela PAM não são enviados para o Workspace.

Conectando PAM e Workspace

Consiste em definir um *listener* no Workspace que detecta perceptos da Memória Perceptual Associativa e os armazena no *Perceptual Buffer* do Workspace. Para isto, definimos um *listener* no Workspace 'escutando' eventos da PAM.

```
<listeners>
    <listener>
      <listenertype>edu.memphis.ccrg.lida.pam.PamListener</listenertype>
      <modulename>PerceptualAssociativeMemory</modulename>
      <listenername>Workspace</listenername>
    </listener>
...
</listeners>
```

A classe PamListener é utilizada pois ela foi implementada no Framework de forma que possa captar eventos específicos da PAM e enviar para algum outro módulo. No nosso caso, definimos que o módulo ouvinte é o Workspace e, mesmo que isso seja óbvio por causa do tipo de *listener* que utilizaremos, precisamos definir a PerceptualAssociativeMemory como sujeito do *listener* por causa do design do Framework.

Neste ponto do exercício, elementos identificados na PAM passam a ser enviados ao Workspace onde ficam armazenados no *PerceptualBuffer.* O exemplo do tutorial já possui uma definição de tarefa na PAM que é capaz de identificar quadrados vermelhos, então quando um destes é inserido no ambiente será percebido pela PAM e agora enviado para o Workspace.

Percebendo elementos

Consiste em definir tarefas que a PAM utiliza com a capacidade de identificar elementos utilizando inputs sensoriais do ambiente. Para isso, definimos tarefas no arquivo que representa *elementFactory* conectando um nome de tarefa a uma classe que implemente a interface *FrameworkTaskImpl* fornecida pelo Framework e associando módulos que podem utilizar esta tarefa. Depois, associamos uma tarefa da PAM a este nome (através da tag taskType) permitindo que a PAM utilize esta classe. Para o caso de detectar círculos vermelhos:

No arquivo referenciado por elementFactory:

```
<task name="ColorDetector">
        <class>myagent.featuredetectors.ColorFeatureDetector</class>
        <ticksperrun>5</ticksperrun>
        <associatedmodule>SensoryMemory</associatedmodule>
        <associatedmodule>PerceptualAssociativeMemory </associatedmodule>
        <param name="color" type="int">-65536</param>
        <param name="node" type="string">red</param>
        </task>
<task name="ShapeDetector">
        <class>myagent.featuredetectors.ShapeFeatureDetector</class>
        <ticksperrun>5</ticksperrun>
        <associatedmodule>SensoryMemory</associatedmodule>
        <associatedmodule>PerceptualAssociativeMemory</associatedmodule>
        <param name="area" type="int">40</param>
        <param name="backgroundColor" type="int">-1</param>
        <param name="node" type="string">square</param>
</task>
```

OBS: As linhas marcadas em vermelho são apresentadas no exercício, porém elas aparentemente não são utilizadas. Estes parâmetros devem ser definidos no momento em que esta tarefa for definida para utilização. Remover estas linhas não muda o comportamento do agente, tampouco são estes parâmetros utilizados como valores *default* como seria de se esperar.

No arquivo referenciado por agentData:

```
<module name="PerceptualAssociativeMemory">
        ...
        <initialTask>
        ...
                <task name="BlueDetector">
                    <tasktype>ColorDetector</tasktype>
```

```
                        <ticksperrun>3</ticksperrun>
                        <param name="color" type="int">-16776961</param>
                        <param name="node" type="string">blue</param>
                    </task>
                    <task name="CircleDetector">
                        <tasktype>ShapeDetector</tasktype>
                        <ticksperrun>3</ticksperrun>
                        <param name="area" type="int">31</param>
                        <param name="backgroundColor" type="int">-1</param>
                        <param name="node" type="string">circle</param>
                    </task>
            ...
            </initialTask>
            ...
        </module>
```

OBS: Podemos ver que são passados parâmetros que são utilizados no momento da identificação do input da memória sensorial. Para a identificação de outras formas e cores, poderíamos utilizar as mesmas definições de tarefas mas com parâmetros diferentes, assim como é feito no *template* do exercício do tutorial para detecção de quadrados vermelhos.

Criando coalisões no Workspace Global utilizando perceptos do *PerceptualBuffer* do Workspace

Consiste em definir tarefas do tipo *AttentionCodelet* no *AttentionModule* do agente que define como a se dá criação de coalisões no *WorkspaceGlobal* quando elementos são inseridos no *PerceptualBuffer.* Em tarefas deste tipo, definimos parâmetros como os nós que devem estar presentes concomitantemente no *PerceptualBuffer* e qual será a ativação inicial da coalisão quando criada e inserida no WorkspaceGlobal.

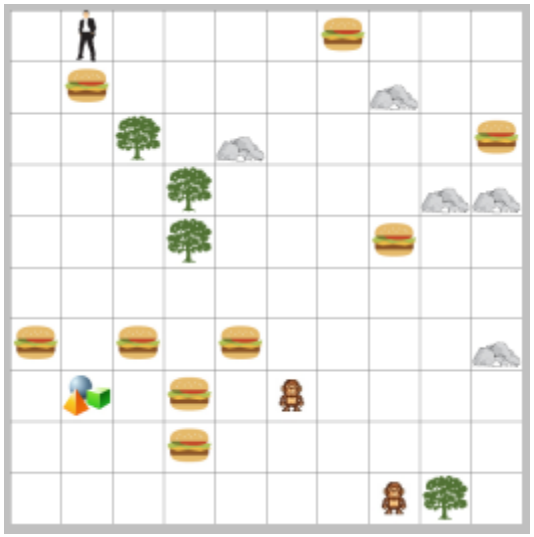
```
<module name="AttentionModule">
        ...
```

```
        <initialTasks>
                ...
                <task name="RedSquareCodelet">
                        <tasktype>BasicAttentionCodelet</tasktype>
                        <ticksperrun>5</ticksperrun>
                        <param name="nodes" type="string">red,square</param>
                        <param name="refractoryPeriod" type="int">30</param>
                        <param name="initialActivation" type="double">1.0</param>
                 </task>
                <task name="BlueCircleCodelet">
                        <tasktype>BasicAttentionCodelet</tasktype>
                        <ticksperrun>5</ticksperrun>
                        <param name="nodes" type="string">blue,circle</param>
                         <param name="refractoryPeriod" type="int">30</param>
                         <param name="initialActivation" type="double">1.0</param>
                        ...
                </task>
        </initialTask>
</module>
```

Atividade 3 – Tutorial Project 2

Esta atividade é baseada na terceira parte do tutorial de exercícios fornecida pelos criadores de LIDA. O tutorial fornece um ambiente virtual simples um agente e outros objetos que serão utilizado para os exercícios.

O Ambiente



Consiste em um tabuleiro formado por células de dimensão 10x10 pixels onde objetos podem existir.

O Agente

Objeto representado pela figura de uma pessoa, possui as seguintes características:

- Um valor de saúde (que se deseja manter elevado). Decresce com o tempo, devido a ataques de macacos, se o agente tentar se mover para fora do tabuleiro ou para uma célula onde exista uma pedra. Aumenta se o agente come um hambúrguer.

- Pode perceber elementos que se encontrem na célula mesma célula que ocupa ou naquela imediatamente afrente

- Pode se deslocar uma célula para frente, mudar a direção que está virado, se alimentar e fugir (se virar e se mover ao mesmo tempo),

- Pode perceber seu próprio valor de saúde.

Predadores

Objetos que representam uma ameaça para o agente. Se movem aleatoriamente pelo tabuleiro e podem tentar machucar o agente se se encontrarem na mesma célula.

Pedras e árvores

Objetos estáticos do ambiente. Não possuem propriedades nem executam ações.

Hambúrgueres

Objetos estáticos do ambiente. Não possuem propriedades nem executam ações. O agente pode consumir este objeto para recarregar sua saúde.

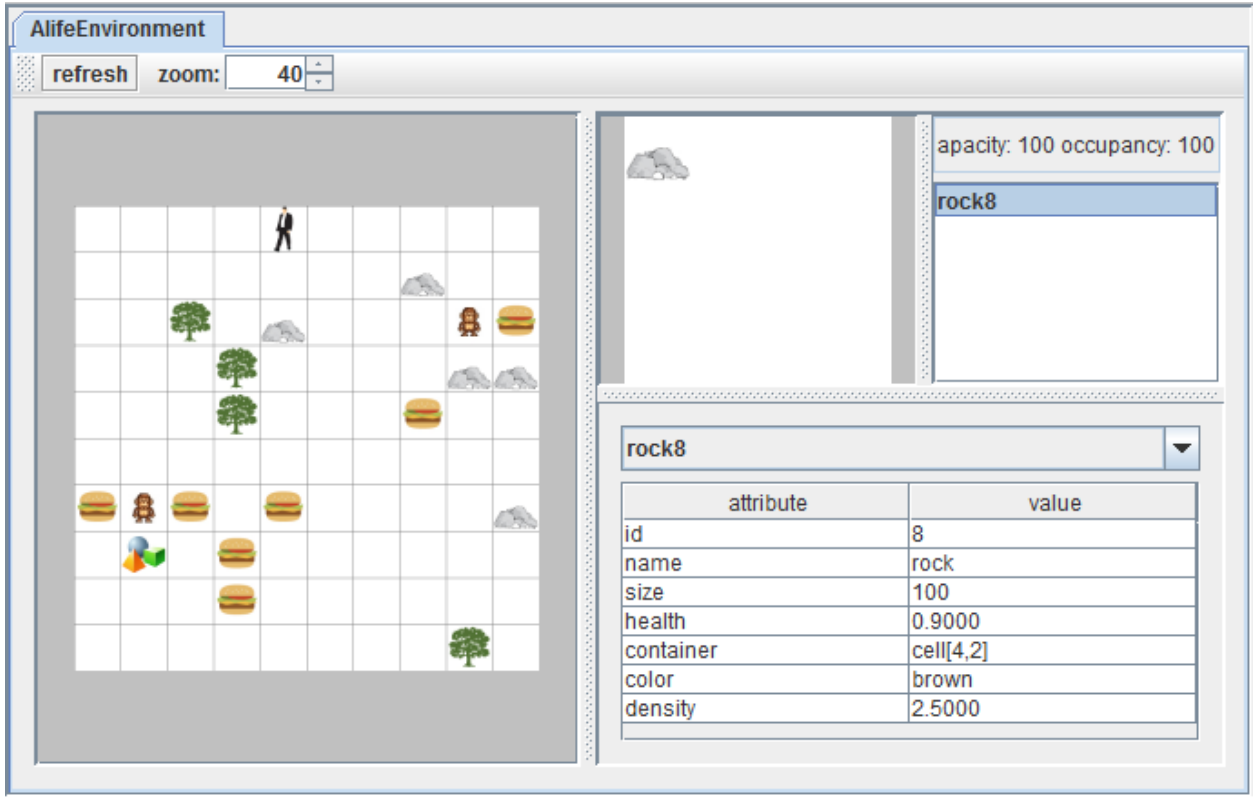
1) Agent Exercise 1

Introdução

O objetivo deste exercício é compreender o funcionamento da interface fornecida pelo projeto do tutorial na interface do simulador e interpretar as informações que esta fornece. Também foi feito um estudo para compreensão do processo para criação do novo painel de monitoramento da simulação.

Painel de monitoramento

Neste projeto, um novo formato de painel foi implementado pelos autores do tutorial, refletindo o novo ambiente e permitindo algumas facilidades para seu monitoramento.



A situação acima ocorreu quando uma das células do ambiente foi selecionada. Os elementos da célula são exibidos nos dois painéis superiores à direita. No primeiro, mais à esquerda, mostra as imagens de todos os elementos. O painel mais à direita mostra o nome das instâncias dos referidos elementos. Neste caso, a célula possui apenas uma pedra. Ao clicarmos no nome em uma das instâncias podemos ver seus atributos no painel inferior. Se uma célula possui mais de um elemento, ela apresenta uma imagem diferenciada que representa esta situação  . Se esta célula for selecionada, haverá mais de uma imagem e mais de uma instância listada nos painéis superiores. Além disso, uma instância pode ser selecionada através do menu *drop down* do painel de atributos.

Para que tudo isso fosse possível, os autores do tutorial seguiram os seguintes passos:

- Foi criado um JPanel Form (ALifeGuiPanel) e modificaram seu código de forma que entendesse a classe GuiPanelImpl (seguindo o padrão do Framework para que o painel possa ser corretamente utilizado)

- Através do arquivo de propriedades do painel (guiPanels.properties), determinaram que o painel carregado pelo framework seria uma instância da classe ALifeGuiPannel

  ```
  environ =AlifeEnvironment,alifeagent.guipanels.ALifeGuiPanel...
  ```

- Dentro deste painel, adicionaram um painel principal (aLifePanel1) e o modificaram de forma que estendesse a classe *ALifePanel* (herdeira da classe *JPanel*) através da janela de customização de código do NetBeans:

  ```
  aLifePanel1 = new edu.memphis.ccrg.alife.gui.ALifePanel();
  ```

  Esta classe está compilada e não temos acesso ao seu código fonte.

- No método *initPanel* da classe ALifeGuiPanel utiliza-se uma classe auxiliar WorldLoader (de código também compilado) para se criar um objeto que implementa a interface ALifeWorldRenderer. Este objeto é então utilizado para se inicializar o objeto aLifePanel1, juntamente com o mundo (resgatado do environment) e um fator de escala.

```
public void initPanel(String[] param) {
     Properties iconsProperties = new Properties();
     environment = (Environment) agent.getSubmodule(ModuleName.Environment);
     if (environment != null) {
          if (world != null) {
               ALifeWorldRenderer renderer =
          WorldLoader.createRenderer(world, iconsProperties);
               if (renderer != null) {
                    ...
                    aLifePanel1.init(renderer, world,scalingFactor);
                    return;
               }
          }
     }
}
```

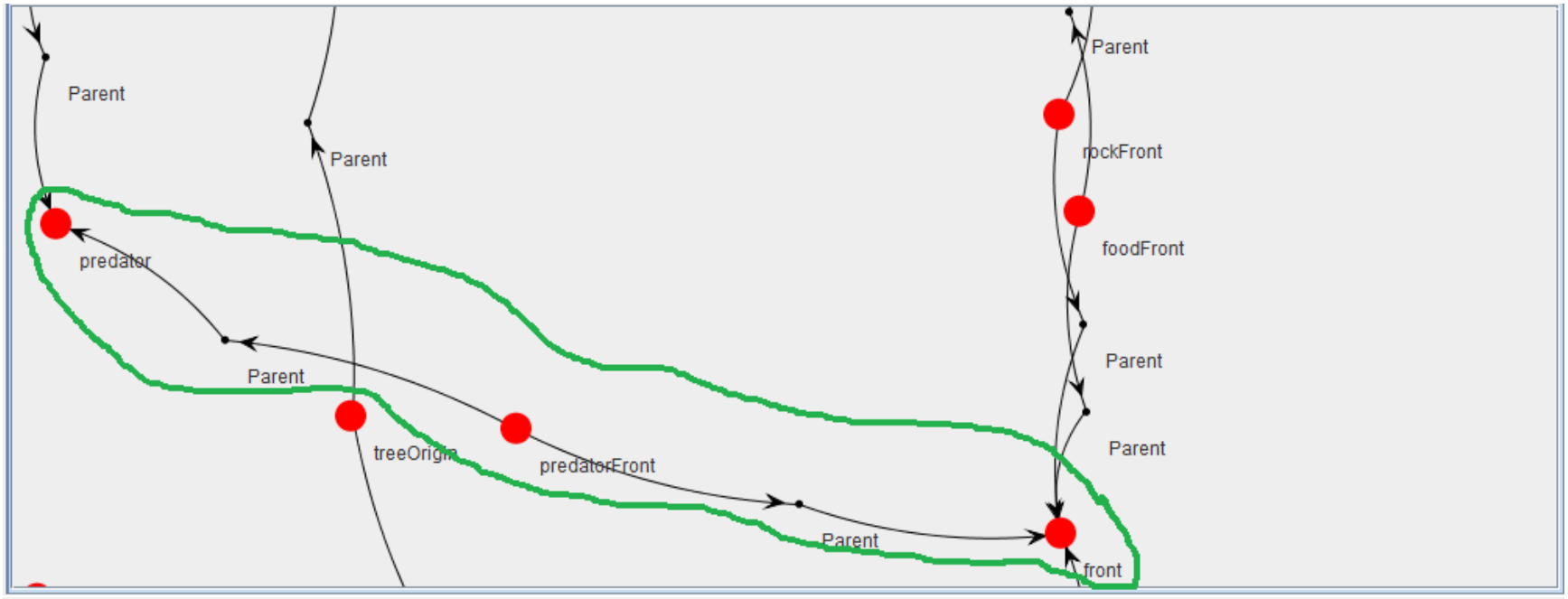
Este processo de criação de um 'renderer' e inicialização do painel ocorre por trás de código compilado portanto não podemos afirmar com certeza o que é feito, mas o resultado final é o painel aLifePanel1 com um link para o objeto mundo do ambiente que deve permitir o resgate de informações conforme necessário de acordo com as ações do usuário (clique em uma célula, seleção de uma instância no menu *drop down*, etc..)

PAM Graph

O grafo da memória associativa perceptual deste exemplo é mais complexo que o dos outros exercícios. Percebe-se claramente que a quantidade de nós e links é muito maior, porém proporcional à quantidade de elementos que podem ser percebidos. Os nós e links que descrevem todo o universo de percepção que o agente tem acesso é definido no xml de definição dos dados do agente (neste caso, aLifeAgent.xml). Neste arquivo, podemos ver nas tags de definição do módulo da PAM a definição de cada nó e em seguida a definição dos links entre eles. Abaixo se encontra o trecho onde são definidos os nós e links e um

pedaço da imagem que representa o grafo da PAM. Os elementos em destaque no XML geram os nós e links em destaque na imagem. Como nos outros exercícios, todos os nós existentes no grafo também são exibidos na tabela PAM Table da GUI do framework.

```
<module name="PerceptualAssociativeMemory">
     ...
     <param name="nodes">
          goodHealth,fairHealth,badHealth,health,
          front,origin,outOfBounds,
          predator,rock,tree,food,
          predatorOrigin,rockOrigin,treeOrigin,foodOrigin,
          predatorFront,rockFront,treeFront,foodFront,emptyFront
     </param>
     <param name="links">
          goodHealth:health,fairHealth:health,badHealth:health,
          rockFront:rock,rockOrigin:rock,
          predatorFront:predator,predatorOrigin:predator,
          foodFront:food,foodOrigin:food,
          treeFront:tree,treeOrigin:tree,
          rockFront:front,predatorFront:front,foodFront:front,treeFront:front,
          rockOrigin:origin,predatorOrigin:origin,foodOrigin:origin,treeOrigin:orig
          in
     </param>
     ...
</module>
```



Execução da simulação

2) Alife Agent Exercise 2

Introdução

Este exercício tem como objetivo ensinar como podem ser criadas tasks para que o agente perceba que está com nível de saúde baixo e que há um inimigo na célula imediatamente a sua frente. Para que isso seja possível, utilizaremos o arquivo aLifeAgent_ex2.xml como arquivo de definição dos dados do agente que não possui as definições necessárias para a execução destas tarefas.

Detectando características do ambiente

Para capacitar o agente a perceber uma característica do ambiente e gerar um nó no grafo da PAM são necessários quatro elementos:

1) Definição de uma tarefa que execute um algoritmo de detecção da característica em questão
2) Carregamento desta tarefa no conjunto de tarefas que o agente pode executar
3) O agendamento da execução desta tarefa no módulo da PAM
4) A definição do nó que deve ser adicionado à PAM quando a detecção ocorrer

Com todos estes elementos definidos, uma tarefa será executada com o algoritmo definido e um nó será gerado na PAM com o nível de ativação retornado pelo algoritmo de detecção.

Percebendo o nível de saúde

Seguindo o procedimento descrito acima, foi criada uma classe *BadHealthDetector* que entende *BasicDetectionAlgorithm*. Esta última, por sua vez, estende *FrameworkTaskImpl* e por isso é considerada uma tarefa. Além disso, *BasicDetectionAlgorithm*, implementa a interface *DetectionAlgorithm* que possui o método *detect*. É na implementação deste método que devemos inserir a lógica de detecção da característica em questão. Tudo o que este método faz é recuperar o valor da saúde do agente através da memória sensorial e comparar com 0.33. Se for maior, a ativação resultante será 0, caso contrário será máxima (1).

```
<module name="PerceptualAssociativeMemory">
     ...
     <initialTasks>
          ...
          <task name="BadHealthDetector">
               <tasktype>BadHealthDetector</tasktype>
               <ticksperrun>3</ticksperrun>
               <param name="node" type="string">badHealth</param>
          </task>
          ...
     </initialTasks>
</module>
```

O segundo passo foi associar esta classe (BadHealthDetector) a uma *task* no arquivo *factoryData.xml*, da seguinte forma:

As tags <associatedmodule> servem para associar módulos com esta tarefa. A classe BasicDetectionAlgorithm possui atributos pamMemory e sensoryMemory. O método **setAssociatedModule** é invocado para cada uma das *tags* presentes na definição da tarefa e, dependendo do tipo de módulo associado, o atributo correspondente receberá o módulo definido na tag após a verificação do tipo do módulo. Seria possível sobrescrever este método para permitir a associação de outros módulos além destes dois. Ao final deste passo, a task BadHealthDetector passa a ficar definida no contexto de execução do agente e pode ser utilizada.

Agora é necessário fazer com que o módulo da PAM utilize esta task. Para isso, basta inserir, no arquivo de configuração dos dados agente (aLifeAgent_ex2.xml) a definição desta task com os parâmetros necessários:

Nota-se a existência do parâmetro “node”. Seu valor é o nó que queremos que esta tarefa gere quando o algoritmo de detecção for executado. No momento de definição da tarefa, este uma referência para um objeto da classe que este nó representa será passado para a tarefa através do método *setPamLinkable* da interface *DetectionAlgorithm*, implementado pela classe *BasicDetectionAlgorithm*.

O último passo seria definir o nó badHealth, mas isto já foi feito e descrito no exercício Agent Exercise 1.

Percebendo um inimigo

Poderíamos seguir os passos anteriores para definir um algoritmo de detecção de um inimigo (macaco) afrente do agente. Contudo, já existe uma classe utilizada por outras tarefas da PAM para detectar objetos. Esta classe, ObjectDetector, foi feita de forma que pode ser utilizada de forma genérica para a detecção de diferentes objetos tanto na célula onde o agente se encontra quanto na célula imediatamente afrente do mesmo.
Um objeto desta classe necessita dois parâmetros principais para detectar um objeto: a posição onde a detecção será feita e o nome do objeto que se está tentando identificar. Se o parâmetro posição for zero, o objeto será buscado na célula onde se encontra o agente. Caso contrário, será na célula imediatamente afrente do agente.
Já existe também a definição de uma *task* associada a esta classe no arquivo *factoryData.xml*

```
<task name="ObjectDetector">
   <class>alifeagent.featuredetectors.ObjectDetector</class>
   <ticksperrun>15</ticksperrun>
   <associatedmodule>SensoryMemory</associatedmodule>
   <associatedmodule>PerceptualAssociativeMemory</associatedmodule>
</task>
```

Com o nó *predatorFront* já definido (AgentExercise 1), tudo o que nos resta fazer é adicionar uma task no módulo da PAM que utilize esta task com os parâmetros necessários para identificação de um predador afrente do agente:

```
<task name="predatorFrontDetector">
        <tasktype>ObjectDetector</tasktype>
        <tickspertun>3</tickspertun>
        <param name="node" type="string"> predatorFront </param>
        <param name="object" type="string"> predator </param>
        <param name="position" type="int"> 1 </param>
</task>
```

Podemos ver que todos os parâmetros necessários para o identificador genérico de objetos estão definidos. O algoritmo irá buscar na célula imediatamente afrente do agente (position = 1) um objeto de com nome predator (object = predator) e o nó criado na PAM será do tipo predatorFront.

Com estas capacidades implementadas podemos ver que o nós são criados na PAM quando o agente tem menos de 3.3 no seu valor de saúde ou se um inimigo se encontra na célula imediatamente afrente.

O resultado da adição destes nós na PAM é que o agente irá se mover quando seu valor de saúde estiver baixo e fugir quando um predador estiver a sua frente. Isto ocorre porque existe uma relação entre a existência destes nós na PAM e uma ação a ser tomada. Este mapeamento de condição – ação é feito no momento da inicialização dos módulos de memória sensorial motora e memória procedural graças à forma como seus *initializers* (*BasicSensoryMotorMemroyInitializer* e *BasicProceduralMemoryInitializer)* são implementados. Basicamente, o *initializer* da memória procedural mapeia um conjunto de nós e links (cuja existência na PAM são a condição para a execução de uma ação) ao nome de uma ação. A memória sensorial motora mapeia o nome de uma ação para uma ação motora do agente (neste caso virtual, um algoritmo ou procedimento). Quando a memória procedural identifica uma condição, o método *processAction* do ambiente é invocado com o nome da ação a ser tomada como parâmetro. Neste momento, o método *performOperation* da classe compilada *ALifeWorld* é invocado e

internamente executa o algoritmo relacionado à ação. Toda esta relação não está clara no tutorial e foi necessária a investigação do código das classes abertas da biblioteca lida.

3) Alife Agent Exercise 3

Introdução

O objetivo deste exercício é a compreensão de como criar modificar parâmetros de um *AttentionCodelet* e qual o efeito destas ações no comportamento de um agente.

Como visto anteriormente, segundo o modelo IDA, a ação que um agente toma em uma iteração do seu ciclo cognitivo é determinada pela 'competição' de coalisões existentes no Workspace Global. Estas coalisões são levadas ao *Workspace Global* pela ação de *Attention Codelets que* buscam no *Perceptual Buffer* do *Workspace* nós e links que satisfaçam a configuração que têm como alvo. Se esta configuração estiver presente, uma coalisão de nós é criada e enviada ao *Workspace Global.* A ação tomada será aquela relacionada com a coalisão mais bem sucedida (com maior nível de ativação) no momento da decisão. O mapeamento de uma coalisão bem sucedida para uma ação é trabalho da Memória Procedural.

Criando um *Attention Codelet*

No contexto do Framework LIDA, um *Attention Codelet* é, como seu próprio funcionamento sugere, uma tarefa. Para criar um novo *attention codelet*, devemos criar uma tarefa num módulo de atenção. Neste projeto, este módulo foi chamado de AttentionModule e utiliza a classe *AttentionCodeletModule*. Para adicionar uma nova tarefa (codelet) ao módulo, utilizamos o arquivo de definição dos dados do agente:

```
<module name="AttentionModule">
     <class>edu.memphis.ccrg.lida.attentioncodelets.AttentionCodeletModule</class>
     <associatedmodule>Workspace</associatedmodule>
     <associatedmodule>GlobalWorkspace</associatedmodule>
     <taskspawner>defaultTS</taskspawner>
     <initialTasks>
          ...
          <task name="predatorAttentionCodelet">
               <tasktype>NeighborhoodAttentionCodelet</tasktype>
               <ticksperrun>5</ticksperrun>
               <param name="nodes" type="string">predator</param>
               <param name="refractoryPeriod" type="int">50</param>
               <param name="initialActivation" type="double">1.0</param>
          </task>
          ...
     </initialTasks>
</module>
```

Foi utilizada a classe *NeighborhoodAttentionCodelet* para a definição do novo codelet. Esta é uma subclasse de *BasicAttentionCodelet* que por sua vez herda AttentionCodeletImpl. A diferença entre *NeighborhoodAttentionCodelet* e *BasicAttentionCodelet* reside no fato de que o segundo cria coalisões contendo apenas os nós que está buscando, enquanto o primeiro cria coalisões com estes nós e todos os seus vizinhos no grafo do *Perceptual Buffer*.

Ativação Inicial

Podemos modificar a ativação inicial de uma coalisão através da modificação do valor da tag de parâmetro *initialActivation*. Como o próprio nome diz, este parâmetro determina a ativação inicial que uma colisão possuirá quando for criada por um *attention codelet.* O nível de ativação da coalisão decai através do tempo de acordo com a estratégia de decaimento utilizada

pelo *Workspace Global. Broadcast Triggers* irão analisar as coalisões existentes no *Workspace* levando em conta suas ativações. Se elas cumprirem a condição de um dos *triggers,* este invocará o *Workspace Global* para que realize a seleção e difusão (*broadcast)* de uma de suas coalisões. A coalisão escolhida é simplesmente aquela com maior nível de ativação.
Se mudarmos, por exemplo, o valor de ativação inicial do *FoodAttentionCodelet* para 0.01 (como sugere o tutorial) perceberemos que dificilmente uma coalisão dos nós *food* serão selecionados pelo *Workspace Global* pois é um nível de ativação muito baixo.

As configurações que regem o comportamento das coalisões são definidas no módulo do Workspace Global, como vê-se a seguir:

```
<module name="GlobalWorkspace">
<class>edu.memphis.ccrg.lida.globalworkspace.GlobalWorkspaceImpl</class>
   <param name="globalWorkspace.coalitionRemovalThreshold" type="double">0.0</param>
   <param name="globalWorkspace.coalitionDecayStrategy">coalitionDecay</param>
   <param name="globalWorkspace.refractoryPeriod" type="int">40 </param>

   <!-- Trigger parameters -->
   <param name="globalWorkspace.delayNoBroadcast" type="int">100 </param>
   <param name="globalWorkspace.delayNoNewCoalition" type="int">50 </param>
   <param name="globalWorkspace.aggregateActivationThreshold" type="double">2.0</param>
   <param name="globalWorkspace.individualActivationThreshold" type="double">0.9</param>
   <taskspawner>defaultTS</taskspawner>

   <initializerclass>edu.memphis.ccrg.lida.globalworkspace.GlobalWorkspaceInitalizer</ini
   tializerclass>
</module>
```

- **coalitionRemovalThreshold**: Determina o menor nível de ativação que uma coalisão pode atingir antes de ser removida do Workspace Global

- **coalitionDecayStrategy:** Determina a estratégia utilizada para decair o nível de ativação das coalisões

- **refractoryPeriod**: Determina a quantidade de *ticks* após um *broadcast* efetuado por um trigger que deve se passar antes que um novo *broadcast* possa ser feito

- **delayNoBroadcast**: Parâmetro utilizado pela instância de *NoBroadcastOcurringTrigger* do *Workspace Global.* O efeito deste parâmetro pode ser entendido como a quantidade suficiente de *ticks* que devem se passar sem que nenhum *broadcast* seja feito por qualquer outro *trigger* para que o *Workspace Global* escolha uma de suas coalisões para difundir pelos módulos do sistema.
  Internamente, é simplesmente o tempo de agendamento de uma *TriggerTask* que a classe *NoBroadcastOcurringTrigger* possui. Sempre que um *broadcast* é efetuado, a classe *GlobalWorkspaceImpl* invoca o método *reset* de cada um de seus *triggers.* Este método, na classe *NoBroadcastOcurringTrigger* reagenda, sua tarefa para depois de **delayNoBroadcast,** de forma que ela nunca será executada enquanto ocorrerem *broadcasts* de outros *triggers* antes que este tempo se passe. Quando esta tarefa é executada, o método *triggerBroadcast* do *Workspace Global* é invocado com a instância de *NoBroadcastOcurringTrigger* como parâmetro o que causa a escolha e difusão de uma coalisão pelo *Workspace Global*.

- **delayNoNewCoalition:** Parâmetro utilizado pela instância de *NoBroadcastOccurringTrigger* do *Workspace Global.* O efeito deste parâmetro pode ser entendido como a quantidade suficiente de *ticks* que devem se passar sem que nenhuma coalisão seja adicionada ao *Workspace Global.* Assim como na classe *NoBroadcastOcurringTrigger*, representa o tempo de agendamento de uma *TriggerTask* desta classe. A diferença entre ambas reside apenas no momento em que o método *reset* é chamado. Sempre que uma nova coalisão é adicionada no *Workspace Global*, todos os *triggers* verificam se devem fazer um *broadcast* (checando a configurações das coalisões do *Workspace Global)*. Esta classe, em vez de fazer qualquer verificação, simplesmente chama seu método *reset*, reagendando sua tarefa para depois de **delayNoNewCoalition** ticks. Assim como no caso de *NoBroadcastOcurringTrigger*, quando a tarefa for executada o *Workspace Global* escolhe e realiza o *broadcast* de uma coalisão.

- **aggregateActivationThreshold:** Parâmetro utilizado pela instância de *AggregateCoalitionActivationTrigger* do *Workspace Global.* Como dito acima, sempre que uma nova coalisão é adicionada ao *Workspace Global*, cada *trigger*

é chamado para verificar o estado do *Workspace* e possivelmente realizar um broadcast. A condição para este trigger é que a soma das ativações das coalisões presentes no *Workspace Global* seja maior ou igual a **aggregateActivationThreshold**. Se isso ocorrer, o Workspace Global realiza a seleção e difusão de uma coalisão pelos módulos do sistema.

- **individualActivationThreshold:** Parâmetro utilizado pela instância de *IndividualCoalitionActivationTrigger* do *Workspace Global.* A condição para este trigger é que alguma das ativações das coalisões presentes no *Workspace Global* seja maior ou igual a **individualActivationThreshold**. Se isso ocorrer, o Workspace Global realiza a seleção e difusão de uma coalisão pelos módulos do sistema.

Attention Codelet Refractory Period

Basicamente, é a frequência com que um *AttentionCodelet* é agendado para execução (vale lembrar: AttentionCodelets são tarefas e portanto são executadas segundo um agendamento). Para alterar o valor do *Refractory Period* de um *AttentionCodelet*, basta mudarmos o parâmetro no arquivo de definição de dados do agente:

```
<module name="AttentionModule">...
     <initialTasks>
          ...
          <task name="GoodHealthAttentionCodelet">
               <tasktype>NeighborhoodAttentionCodelet</tasktype>
               <ticksperrun>5</ticksperrun>
               <param name="nodes" type="string">goodHealth</param>
               <param name="refractoryPeriod" type="int">50</param>
               <param name="initialActivation" type="double">0.10</param>
          </task>
        ...
     </initialTasks>
</module>
```

O tutorial sugere que alteremos o valor do *refractoryPeriod* do *GoodHealthAttentionCodelet* para 10 e verifiquemos o efeito disso no comportamento do agente. Como diminuir este valor significa executar esta tarefa mais vezes, coalisões com o nó *goodHealth* e todos os seus vizinhos (*health*) serão mais frequentemente adicionados ao *Workspace Global* e é justamente o que se observa: enquanto o agente possui mais que 0.66 em seu valor de saúde basicamente sempre é observada a presença de uma coalisão dos nós *health* e *goodHealth.*

4) Alife Agent Exercise 4

Introdução

Este exercício tem como objetivo ilustrar como configurar a ação que um agente executa quando uma determinada coalisão é difundida através do sistema, como utilizar *Initializers* customizados para módulos do sistema, como adicionar links dinamicamente entre nós da PAM utilizando estes *Initializers* e como modificar a estratégia de decaimento da ativação de nós da PAM.

Definindo uma ação para uma coalisão difundida através do sistema

Quando o *Workspace Global* seleciona uma coalisão ele itera por todos os módulos registrados como *listeners* chamando o método *receiveBroadcast* com uma cópia do conteúdo da coalisão como parâmetro. Devido a este *broadcast,* outros módulos irão agir de forma que uma ação será selecionada e executada.

Em especial, um dos módulos registrados no *Workspace Global* é a Memória Procedural. A memória procedural contém definições dos chamados **esquemas**. A cada broadcast que recebe, este módulo pode criar instâncias de seus esquemas (uma instância de esquema é chamado comportamento) e enviá-lo para o módulo de seleção de ação do sistema (neste caso, o módulo utilizado é da classe BasicActionSelection). Um esquema possui um contexto, uma ação e um resultado. A relação entre estes elementos é, basicamente, a de que dado um contexto, tomar determinada ação resultará em um resultado.

A seleção de ação se baseia no nível de ativação de cada *behavior* (o de maior ativação é escolhido) e o *behavior* escolhido é então finalmente enviado para o módulo da Memória Sensorial-Motora, que determina o algoritmo que deve ser executado para efetuar a ação do *behavior*.

Os parâmetros envolvidos são definidos através do arquivo de definição dos dados do agente, como se segue nos trechos incompletos a seguir:

```
<module name="ActionSelection">
    <class>edu.memphis.ccrg.lida.actionselection.BasicActionSelection</class>
    <param name="actionSelection.ticksPerStep" type="int"> 10</param>
    <taskspawner>defaultTS</taskspawner>
</module>

<module name="SensoryMotorMemory">
    ...
    <param name="smm.3">action.turnAround,algorithm.turnAround</param>
    <param name="smm.4">action.moveAgent,algorithm.moveAgent</param>
    ...
</module>


<module name="ProceduralMemory">
    ...
    <param name="scheme.10">if emptyFront turn left| (emptyFront)()| action.turnLeft|
()()| 0.1</param>
    <param name="scheme.10a">if emptyFront turn right| (emptyFront)()|
action.turnRight| ()()| 0.1</param>
    <param name="scheme.10b">if emptyFront turn around| (emptyFront)()|
action.turnAround| ()()  |0.1</param>
    ...
</module>
```

Foram destacados em vermelho partes relacionadas no fluxo de seleção de ação.
O primeiro conjunto de tags caracteriza o módulo de seleção de ação e a classe que será utilizada.

Na parte da memória procedural, é definido um esquema de nome “scheme.10b”. A primeira parte do esquema (if emptyFront turn around) é uma descrição em linguagem comum (que não é utilizada para efeitos de execução) do esquema. A segunda

parte define os nós e links que representam o contexto. Neste caso, a existência do nó *emptyFront* é em uma coalisão difundida deve disparar a ação deste esquema. A terceira parte descreve o nome da ação que deve ser tomada (action.turnAround). A última parte define os nós e links que representam o resultado esperado da tomada desta ação e para esta aplicação foi deixada sem elementos.

A parte da memória sensorial-motora define que algoritmos deve ser executados para realizar diferentes ações. Neste caso, a ação ***action.turnAround*** está mapeada para ***algoritm.turnAround***. Esta é uma chave que leva a um método específico. Este mapeamento se encontra no arquivo operations.properties, que é utilizado no momento de inicialização do módulo da memória sensorial-motora e associa as ações, criadas pela memória procedural, nome completo do método que deve ser utilizado para aquela ação. O trecho do arquivo ***operations.properties*** que possui estas propriedades é exibido abaixo:

```
algorithm.turnLeft=alifeagent.environment.operations.TurnLeftOperation
algorithm.turnRight=alifeagent.environment.operations.TurnRightOperation
algorithm.turnAround=alifeagent.environment.operations.TurnAroundOperation
algorithm.moveAgent=alifeagent.environment.operations.MoveAgentOperation
algorithm.eat=alifeagent.environment.operations.EatOperation
algorithm.flee=alifeagent.environment.operations.FleeOperation
```

O tutorial propõe que mudemos a ação do esquema 10b de forma a permitir que o agente se mova se não houver nada em sua frente.

```
<param name="scheme.10b">if emptyFront turn around|(emptyFront)()|action.moveAgent
|()()  |0.1</param>
```

Utilizando um *Initializer* customizado

Consiste em modificar, no arquivo de configuração de dados do agente, a classe que é utilizada para inicializar um módulo do sistema. O tutorial sugere que alteremos o *Initializer* do módulo da PAM. Para isso, basta mudar o nome da classe na tag *initializerclass* na configuração deste módulo. No caso, utilizaremos a classe **alifeagent.initializers.CustomPamInitializer**

```
<module name="PerceptualAssociativeMemory">
     <class>edu.memphis.ccrg.lida.pam.PerceptualAssociativeMemoryImpl</class>
     <param name="pam.Upscale" type="double">.7 </param>
     ...
     <taskspawner>defaultTS</taskspawner>
     <initialTasks>
        ...
     </initialTasks>
         <initializerclass>alifeagent.initializers.CustomPamInitializer
         </initializerclass>
</module>
```

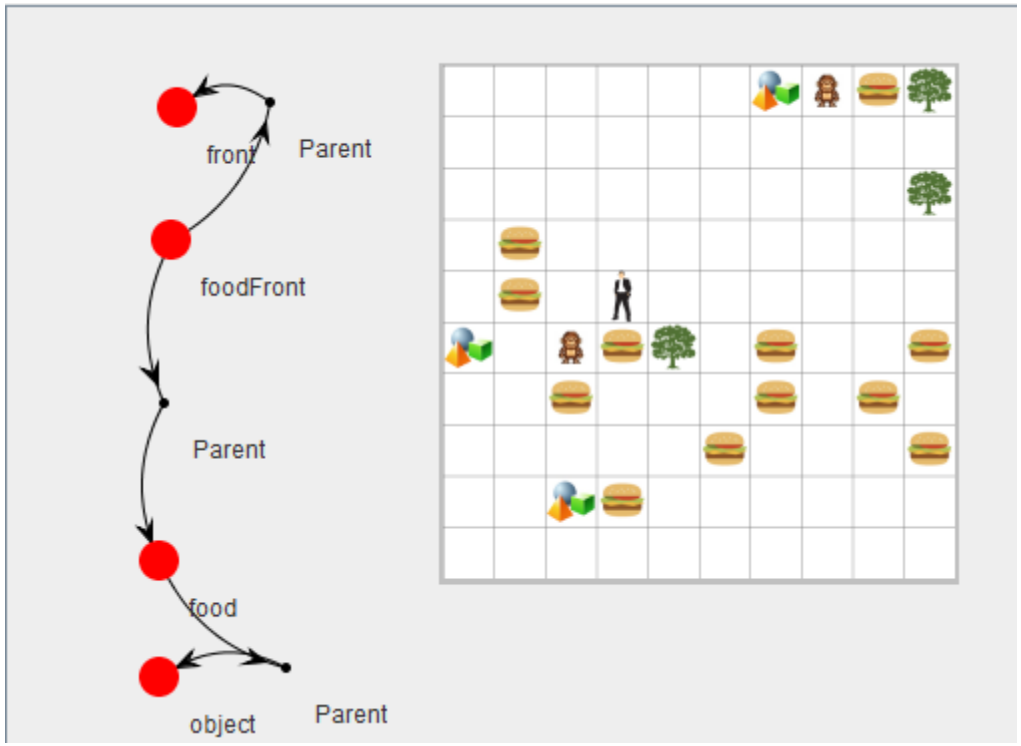
Ao executarmos o agente novamente, podemos verificar que o método *initModule* desta classe é invocado.

O tutorial propõe que esta classe seja utilizada para adicionar um novo *link* entre o nó *food* e o nó *object* (que é criado nesta classe sem a utilização do arquivo de configuração de dados do agente). Para isso, basta recuperarmos uma referência para o nó *object* (criado neste *initializer*), outra para o nó *food* (criado na superclasse utilizando o arquivo de configurações de dados do agente) e criar, através do método *addDefaultLink* da PAM, um link entre eles:

```
child = pam.getNode("food");

pam.addDefaultLink(factory.getLink(child, objectNode,
PerceptualAssociativeMemoryImpl.PARENT));
```

Desta forma, sempre que o nó *food* inserido no *Perceptual Buffer*, o nó object será 'carregado' porque existe um link entre eles. A imagem abaixo ilustra a situação do Perceptual Buffer quando há um hambúrguer na célula imediatamente afrente do agente:



Alterando a estratégia de decaimento de um nó

Como visto anteriormente, a ativação de um nó decai conforme o tempo passa de acordo com uma estratégia definida para aquele nó. O tutorial propõe que alteremos a estratégia de decaimento do nó *object* de forma que ele demore mais para ser removido do *Perceptual Buffer*. O trecho de código abaixo ilustra como definir para o nó *object* a estratégia *slowDecay.*

```
Node objectNode = pam.addDefaultNode(factory.getNode("PamNodeImpl", "object"));
DecayStrategy decayStrategy = factory.getDecayStrategy("slowDecay");
objectNode.setDecayStrategy(decayStrategy);
```

Com esta alteração fica evidente que o nó *object* permanece com sua ativação igual a 1 por muito mais tempo mesmo quando os nós *rock* e *food* já tenham sido removidos devido seu decaimento.

5) Advanced Exercise 1

O tutorial propõe a adição de um *attentionCodelet* e mapeamento na memória procedural para que o agente se mova para frente se houver uma árvore nesta posição. Para isto, basta adicionarmos um novo *attentionCodelet* para que o nó *treeFront* seja movido para o *Workspace Global* e em seguida um esquema na memória procedural para que a presença deste dispare a ação *moveAgent*. Isso foi facilmente feito adicionando os seguintes trechos no arquivo de configuração de dados do agente:

```
...
<module name="ProceduralMemory">
    ...
    <param name="scheme.11">if treeFront move
agent|(treeFront)()|action.moveAgent|()()  |0.1</param>
    ...
</module>
...
<module name="AttentionModule">
    ...
    <initialTasks>
        ...
        <task name="TreeAttentionCodelet">
            <tasktype>NeighborhoodAttentionCodelet</tasktype>
            <ticksperrun>5</ticksperrun>
            <param name="nodes" type="string">treeFront</param>
            <param name="refractoryPeriod" type="int">50</param>
            <param name="initialActivation" type="double">1.0</param>
        </task>
    </initialTasks>
</module>
...
```

6) Advanced Exercise 2

Este exercício tem como objetivo alterar a forma que o sistema do agente seleciona ações a tomar a cada ciclo cognitivo. Para isso, se vê necessária a implementação de uma módulo de seleção de ação customizado e sua utilização pelo sistema. A estratégia utilizada para a execução do exercício é de utilizar a classe *BasicActionSelection* fornecida pela *Framework* com a modificação do método utilizado para selecionar um *Behavior* e tratamento do recebimento de *Broadcasts* provenientes do

Workspace Global. O ideal seria estender a classe *BasicActionSelection*, mas como grande parte de seus métodos e propriedades são privados (impedindo o acesso através de subclasses) se fez necessário uma cópia de seu código na íntegra.

O tutorial sugere que a lógica de seleção escolha apenas *Behaviors* cujos nós do contexto se encontrem no último *Broadcast* recebido pelo seletor. Em vez disso, foi escolhido o uso do cálculo percentual da existência dos nós do contexto do *Behavior* no último *Broadcast* para decidir o mais eficiente para o contexto atual. Abaixo segue o trecho de código da classe *BasicActionSelection* responsável por escolher o *Behavior* no ciclo cognitivo e receber Broadcasts:

```
public class BasicActionSelection extends FrameworkModuleImpl implements
          ActionSelection, ProceduralMemoryListener {

     private Behavior chooseBehavior() {
          Behavior selected = null;
          double highestActivation = -1.0;
          for (Behavior b : behaviors) {
               double behaviorActivation = b.getActivation();
               if (
                    behaviorActivation >= candidateThreshold
                    && behaviorActivation > highestActivation)
               {
                    selected = b;
                    highestActivation = behaviorActivation;
               }
          }
          return selected;
     }

     @Override
     public void receiveBroadcast(BroadcastContent bc) {
     }
}
```

Enquanto o método de tratamento de Broadcasts não faz nada, o método *chooseBehavior* escolhe dentre os Behaviors aquele que possui maior ativação e que tem ativação maior que um mínimo estabelecido (*candidateThreshold*).
Abaixo se encontra todo o código adicionado ou modificado (destacado em verde) na classe customizada MyBasicActionSelection. Todo o restante do código é exatamente igual ao da classe original:

```
public class MyBasicActionSelection extends FrameworkModuleImpl implements ActionSelection,
ProceduralMemoryListener {
	...
	private static final double DEFAULT_MINIMUN_BEHAVIOR_NODES_PRESENCE = 0.75;
	private double minimunBehaviorNodesPresence;
	...

	@Override
	public void init() {
		...
		minimunBehaviorNodesPresence = (Double)
		getParam("actionSelection.minimunBehaviorNodesPresence",
		DEFAULT_MINIMUN_BEHAVIOR_NODES_PRESENCE);
		...
	}
```

```
	@Override
	public void receiveBroadcast(BroadcastContent bc) {
		NodeStructure bcStructure = (NodeStructure) bc;
		this.lastBroadcast = bcStructure;
	}

	private float getNodePresenceOnLastBroadcast(Behavior b){
		int presentNodes = 0;
		for(Node node : b.getContextNodes()){
			if(this.lastBroadcast.containsNode(node)){
				presentNodes++;
			}
		}
		float percentagePresent = presentNodes/b.getContextNodes().size();
		return percentagePresent;
	 }

	private Behavior chooseBehavior() {
		Behavior selected = null;
		double highestActivation = -1.0;
		for (Behavior b : behaviors) {
			double behaviorActivation = b.getActivation();
			double nodePresenceOnLastBroadcast = getNodePresenceOnLastBroadcast(b);
			if (behaviorActivation >= candidateThreshold &&
				nodePresenceOnLastBroadcast >= minimunBehaviorNodesPresence &&
				behaviorActivation + nodePresenceOnLastBroadcast > highestActivation)
			{
				selected = b;
				highestActivation = behaviorActivation + nodePresenceOnLastBroadcast;
			}
		}
		return selected;
	}
}
```

Sempre que recebe um *Broadcast*, o seletor guarda a estrutura de nós recebida. No momento da seleção, ele utiliza estes dados para utilizar no método *chooseBehavior*.

Abaixo são exibidos os trechos de código alterados no arquivo de configuração de dados do agente para permitir a utilização da classe customizada desta forma.

```
<lida>
      ...
    <submodules>
        ...
        <module name="ActionSelection">
            <class>alifeagent.modules.MyBasicActionSelection</class>
            <param name="actionSelection.ticksPerStep" type="int"> 10</param>
            <param name="actionSelection.minimunBehaviorNodesPresence"
type="double">0.7</param>
            <taskspawner>defaultTS</taskspawner>
        </module>
        ...
        </module>
    </submodules>
    ...
    <listeners>
        ...
        <listener>
        ...
            <modulename>GlobalWorkspace</modulename>
            <listenername>ActionSelection</listenername>
        </listener>
        ...
    </listeners>
</lida>
```