Relatorio 02 - SOAR: Controlando o WorldServer3D

Leonardo de Oliveira Ramos RA171941

Abril 2018

1 Como executar

Apenas abra a pasta run e rode o script run.sh com o comando "sh diretorio/run.sh"

2 Atividade 1

2.1 Main.java

2.1.1 NativeUtils e outras classes

No código da Main.java encontra-se o uso da classe NativeUtils para carregar o arquivo soar-rules.soar do arquivo jar da compilação. Sendo o arquivo soar-rules.soar o arquivo que contém as decisões reativas para demonstração.

Em seguida são declaradas as variáveis das classes Environment, MindView e SoarBridge que serão futuramente melhor explicadas.

2.1.2 Loop Principal

O loop principal é um loop infinito (while(true)) que só termina quando o programa é fechado. Ele começa verificando se a variável do MindView, debugState é 0, caso seja, o método SoarBridge.step() é chamado, rodando um loop inteiro do SOAR, processando já o input link (parseando a string e dando os comandos).

Em seguida o input link e output link são mandados para a interface gráfica imprimir.

2.2 SoarBridge.java

2.2.1 Métodos step() e mstep()

O método step() serve para rodar o SOAR, para isso ele coloca as informações na string do input link através do método prepareInputLink() em seguida de stringInputLink(), roda o SOAR por um passo completo, recebe uma string do output link com os comandos, parseia e executa os comandos usando o CommandUtility do WS3DProxy.

O mstep() tem o mesmo conceito porém ele só roda um micro passo, ou seja, roda apenas uma das fases do passo do SOAR.

2.3 Leitura do estado do ambiente no WorldServer3D

A leitura do estado do ambiente é feita pelo prepareInputLink() que através de informações do Environment, recebe a instancia da criatura e seus parâmetros e visões. Então é preparada uma string que contém informações para colocar no input link.

2.4 soar-rules.soar

Estas regras apenas olham as entidades (joias e comidas) mais próximas e anda em direção a elas.

2.4.1 Wander

Primeiramente ele define o operador wander, que define as velocidades das rodas do jogador fazendo ele rodar (velocidade em apenas uma roda). O operador wander tem prioridade baixa, dado que ele fica em wander somente se não tiver nada de útil para fazer.

2.4.2 Memory operators para COUNT

Depois ele define o que são as memórias usadas, elas são para contar o número de coisas na sacola do robo (que vem pelo input-link).

2.4.3 Andar e pegar joias e comida

Então ele define o operador que anda com o robô em direção a uma joia e para comida, e também os operadores que percebem quando a comida ou joia está perto para poder pega-la.

2.4.4 AvoidBrick

Em seguida tem o avoidBrick que serve para desviar se houver um obstáculo na frente.

2.4.5 Impasse solving e preferências

OPERATORS PREFERENCES AND IMPASSE SOLVE são para definir quais serão os operadores escolhidos caso haja duvida entre 2 operadores.

2.4.6 Halt

Halt condition é para definir quando deve-se interromper o fluxo e começar de novo.

2.4.7 Comportamento reativo do soar-rules.soar

É notável dizer que o comportamento deste programa é reativo, isso significa que ele tem uma reação direta a cada input, sem analisar a árvore de decisões, apenas vai olhando quais jóias estão mais perto para pegar.

3 Atividade 2

3.1 DemoSoar

As principais mudanças feitas no DemoSoar foram principalmente no arquivo SoarBridge.java, em que adicionei um código para colocar no input-link os valores de todas as entidades do WorldServer (nomeado WORLD) e para mandar as cores de cada joia do leaflet que falta pegar (OBJECTIVE).

O código alterado para o envio dos dados para o input-link:

```
worldEntities = CreateIdWME(inputLink, "WORLD");
            List<Thing> thingsListWorld = w.getWorldEntities();
            for (Thing t : thingsListWorld) {
                Identifier entity = CreateIdWME(worldEntities, "ENTITY");
                CreateFloatWME(entity, "DISTANCE", GetGeometricDistanceToCreature(t
                CreateFloatWME(entity, "X", t.getX1());
                CreateFloatWME(entity, "Y", t.getY1());
                CreateFloatWME(entity, "X2", t.getX2());
                CreateFloatWME(entity, "Y2", t.getY2());
                CreateStringWME(entity, "TYPE", getItemType(t.getCategory()));
                CreateStringWME(entity, "NAME", t.getName());
                CreateStringWME(entity, "COLOR", Constants.getColorName(t.getMateri
            }
            objectives = CreateIdWME(inputLink, "OBJECTIVE");
            List<Leaflet> leaflets = c.getLeaflets();
            float leaf_number = 0;
            for (Leaflet l : leaflets) {
                Identifier leaf = CreateIdWME(objectives, "LEAFLET");
                CreateFloatWME(leaf, "NUMBER", leaf_number);
                leaf_number += 1;
                HashMap<String, Integer[]> items = null;
                items = l.getItems();
                for (String str : items.keySet()) {
                    //jewel color
                    int values = l.getMissingNumberOfType(str);
                    for(int i = 0; i < values; i++)
                    {
                        Identifier color = CreateIdWME(leaf, "COLOR");
                        CreateStringWME(color, "VALUE", str);
                        CreateFloatWME(color, "NUMBER", leaf_number);
                        leaf_number += 1;
                    }

                }

            }
        }
```

3.2 soar-deliberative.soar

Dado este input-link, eu precisei copiar os dados para a memória interna para poder altera-la, para isso existe uma etapa que que inicializa os valores do input-link: as entidades do mundo vão para WORLD, e o leaflet vai para OBJECTIVES, que contém os 3 leaflets, que contém uma estrutura chamada COLOR que tem o valor da cor em VALUE e o id dela em NUMBER, e mais tarde terá a leaflet selecionada para essa cor do leaflet.

Depois tem uma etapa de decidir qual joia vai para qual leaflet, para isso eu analiso para cada cor, ele escolhe a mais próxima.

Em seguida é definido as funções de ir até a joia e pegar a jóia, ir até a comida e comer a comida. Dados estes operadores tem-se que resolver qual tem preferência pelo impasse tie, assim está definido que:

- Colocar na lista de planos a ação de andar até jóia que está mais perto primeiro.
- Colocar na lista de planos a ação de pegar uma jóia tem preferência a andar até uma jóia.
- Colocar na lista de planos a ação de ir até a comida tem preferência quando a energia está abaixo de um threshold.
- Comer tem preferência a todos, executando reativamente

Então quando o robô tem algo para fazer (pegar joia ou comida) ele faz isso, mas quando ele não tem nada para fazer ele fica parado, esperando um novo spawn de joias, e se ele já pegou todas as joias e não está com fome, mesmo assim ele vai atrás de comida.

A etapa de planning segue analisando as joias escolhidas e cria uma estrutura similar a de um vetor dentro de PLAN no estado principal. Esse plano é criado diretamente de acordo com o parâmetro DISTANCE das entidades, decidindo a ordem de quais entidades ele escolherá, comida ou joia, qual estiver mais perto.

3.3 Comparando antigas versões e a final

Primeiramente eu fiz experimentos de colocar 2 soar_deliberative juntos, eles sempre acabavam indo para as mesmas joias, sempre competindo, e quando ficavam muito perto eles travavam (colidiam).

Colocando o soar_deliberative com outros mais antigos como eles não competiam tanto, eles acabavam demorando mais para colidir, e como o soar_deliberative não pega joias inúteis ele acabava ganhando. Uma nota interessante seria que como ele não vai atrás de comida quando estava sem fome, ele chegou a ficar sem energia pois o outro tinha pego todas as comidas.