UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI
FACULDADE DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE COMPUTAÇÃO

O MUNDO DO WUMPUS EM LEGO-MINDSTORM E PROLOG

Christopher Albert Ferreira de Aguiar

Diamantina
2016

UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI
FACULDADE DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE COMPUTAÇÃO

O MUNDO DO WUMPUS EM LEGO-MINDSTORM E PROLOG

Christopher Albert Ferreira de Aguiar

Orientador:

Cristiano Grijó Pitangui

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação, como parte dos requisitos exigidos para a conclusão do curso.

Diamantina

2016

AGRADECIMENTOS

A minha família, amigos e colegas de curso, por todo o incentivo que me deram para superar as muitas dificuldades. A esta universidade, seu corpo docente, direção e administração que me deram a oportunidade de buscar novos caminhos e visualizar um novo horizonte.

Ao meu orientador Cristiano, pelo tempo que dedicou para me oferecer suporte, pelas suas correções e incentivos.

A todos que de maneira direta ou indireta, fizeram parte da minha formação, muito obrigado.

RESUMO

A Inteligência Artificial desenvolve sistemas computacionais com comportamento inteligente. O projeto de sistemas inteligentes passa por algumas etapas e uma das mais importantes é a escolha do tipo de agente racional. A Inteligência Artificial apresenta tipos básicos de agente como o agente reativo simples e o agente baseado em modelos. Escolher o modelo de agente racional correto pode determinar o sucesso ou fracasso do projeto. O Mundo de Wumpus é um problema que serve como ambiente de aprendizado e testes de agentes racionais.

O conhecimento do agente racional deve possuir uma estrutura bem definida e flexível, permitindo ao o agente gerar conhecimento a partir do que ele já sabe, utilizando um mecanismo de inferência. Assim, é preciso definir uma base de conhecimento através de uma linguagem de representação de conhecimento. Uma das linguagens mais utilizadas é a Lógica de Predicados, ela possui elementos que facilitam a definição tanto de bases de conhecimento como de mecanismos de inferência. Para que a base de conhecimento possa fazer parte de um programa de agente, é necessária uma linguagem de programação capaz de representar toda a lógica envolvida. Prolog é uma linguagem de programação lógica e possui características que facilitam essa tarefa.

O Kit LEGO Mindstorms, surgiu como uma ferramenta para desenvolver o raciocínio e criatividade de crianças e jovens, porém devido ao grande sucesso, suas funcionalidades foram expandidas, e hoje é utilizado em disciplinas de introdução à programação de computadores, aprendizado e até mesmo em campeonatos de robótica. Existem várias interfaces alternativas ao *software* fornecido pelo fabricante, para aqueles que desejam tirar maior proveito das capacidades do kit, as mais comuns são escritas em C/C++. Neste trabalho foi utilizada a interface NXT++ escrita em C++.

O escopo deste trabalho é criar um agente racional físico, no caso um robô e um computador, utilizando C++, Prolog, Lógica de Predicados e o kit LEGO Mindstorms para solucionar uma adaptação do Mundo de Wumpus para um ambiente real.

Palavras-chave: Inteligência Artificial, agentes racionais, mundo de Wumpus, Lógica de Predicados, Prolog, C++, SWI-Prolog.

ABSTRACT

Artificial Intelligence develops computer systems with intelligent behavior. The intelligent system design goes through some steps and one of the most important is the choice of the rational agent type. Artificial Intelligence has basic types of agents as the simple reactive agent and agent-based models. Choosing the correct rational agent model can determine the success or failure of the project. The Wumpus World is a problem that serves as a learning environment and rational agents tests.

The knowledge of rational agent should have a well-defined and flexible structure, allowing the agent to generate knowledge from what he already knows, using an inference engine. Thus, it is necessary to define a knowledge base through a knowledge representation language. One of the most used languages is the Predicate Logic; it has elements that facilitate the definition of both knowledge bases and inference mechanisms. The knowledge base can be part of an agent program and a programming language capable of representing all of the logic involved is necessary. Prolog is a logic programming language and has features that facilitate this task.

The kit LEGO Mindstorms has emerged as a tool to develop the creativity of children and young people, but due to the great success, its functions were expanded, and now it is used in introductory courses in computer programming, learning and even robotics championships. There are several alternative interfaces to the software provided by the manufacturer, for those wishing to take full advantage of the kit's capabilities, the most common are written in C/C ++. In this work we used the NXT ++ interface written in C ++.

The scope of this work is to create a physical rational agent, in this case a robot and a computer, using C ++, Prolog, Predicate Logic and the LEGO Mindstorms kit to explore a Wumpus World adaptation to a real environment.

Keywords: Artificial Intelligence, rational agents, Wumpus World, Predicate Logic, Prolog, C ++, SWI-Prolog.

Sumário

1 INTRODUÇÃO

A Inteligência Artificial está associada ao conceito de conhecimento. O computador pode utilizar o conhecimento fornecido por projetistas para a solução de problemas. Os sistemas inteligentes abrangem, por exemplo, determinados robôs que demonstram comportamento inteligente.

Um sistema de Inteligência Artificial não é capaz somente de armazenamento e manipulação de dados, mas também da aquisição, representação, e manipulação de conhecimento, com capacidade para deduzir ou inferir novos conhecimentos - novas relações sobre fatos e conceitos a partir do conhecimento existente, para solução de problemas complexos. Há muitas questões a serem contornadas pela engenharia do conhecimento no projeto do agente inteligentes, por exemplo: aquisição, representação e manipulação de conhecimento, um mecanismo de inferência que determina os itens de conhecimento a serem acessados, as inferências a serem feitas, quais passos a serem executados e a ordem da execução.

O conceito de agentes inteligentes serve tanto para o aprendizado quanto para auxiliar o projetista na construção de um agente completo. Passando pela definição de atuadores e sensores, da medida de desempenho e as regras de inferência. Os tipos básicos de agente inteligente e suas variações permitem escolher aqueles que mais se adequam ao problema que se pretende resolver.

Um exemplo de aplicação do conceito de agente inteligente em projetos de sistemas de inteligência artificial é o mundo de Wumpus. Por ser simples, tal problema serve como ambiente de aprendizagem para problemas de Inteligência Artificial, contudo, é mais comumente explorado apenas no ambiente puramente virtual, ou seja, apenas em software.

O kit LEGO Mindstorms foi criado inicialmente para auxiliar no desenvolvimento intelectual de crianças e jovens, porém passou a ser utilizado como ferramenta de aprendizado em inteligência artificial. Neste projeto não foi diferente, o kit foi utilizado para criar uma estrutura física, um robô, que fosse capaz de explorar um ambiente com as configurações do ambiente de tarefas do mundo de Wumpus, agir sobre ele e assim solucionar o problema.

A estrutura física não é o suficiente, é preciso que o robô seja capaz de raciocinar para encontrar uma solução. O agente deve possuir uma forma de inferir fatos sobre a porção

não observável do ambiente. Dessa forma, é preciso criar uma base de conhecimento que é responsável por representar o conhecimento do agente a respeito do ambiente, além de permitir que ele faça as inferências necessárias.

Para implementar a base conhecimento é preciso uma linguagem de representação de conhecimento. Prolog é uma linguagem de programação que tem como propósito principal, representar o conhecimento sobre o mundo real ou virtual. Por seguir o paradigma de programação Lógica, Prolog fornece as ferramentas básicas para implementar uma base de conhecimento, bem como o mecanismo de inferência. Contudo, Prolog não possui as características necessárias para lidar com todo tipo de dispositivo. Sendo assim, C++ é a linguagem de programação mais indicada, pois a maioria dos *firmwares* dos dispositivos, sistemas operacionais e drivers são escritos em C/C++.

Para unir as duas linguagens é preciso uma interface entre as duas, para isto, utilizou-se o SWI-Prolog que é um compilador/interpretador de Prolog que fornece uma interface bem definida, porém com documentação muito resumida e escassa.

1.1 Objetivo Geral

Este trabalho trata do aprendizado e aplicação prática de vários conceitos relativos à Inteligência Artificial e programação tais como: o que é um agente inteligente e quais os principais tipos de agente, Lógica de Predicados, Prolog e C++ e a interface SWI-Prolog para C++.

1.2 Objetivos Específicos

1. Resolver o problema do mundo de Wumpus em um ambiente real, utilizando o kit LEGO Mindstorms, Lógica de Predicados, Prolog e C++.
2. Servir como documentação introdutória para programadores que precisam unir C++ e Prolog no mesmo programa.

1.3 Organização do Trabalho

- **O Capítulo 2** apresenta o conceito de agentes inteligentes, suas principais características e os principais tipos básicos de agente. Aborda o conceito de ambiente de tarefas, os principais tipos de ambiente e suas características.
- **O Capítulo 3** apresenta o mundo de Wumpus, suas regras e classifica o seu ambiente de tarefas de acordo com suas características.
- **O Capítulo 4** apresenta as principais ferramentas utilizadas, mostra detalhes da definição da base de conhecimento e do programa em C++. Aborda também, como foi possível unir Prolog e C++ e demonstra como foi possível controlar o robô LEGO via Bluetooth utilizando um computador.
- **O Capítulo 5** apresenta como foram realizados os testes, uma execução passo-a-passo, as principais dificuldades encontradas, os resultados finais e discussão sobre os testes.
- **O Capítulo 6** apresenta as conclusões obtidas a partir de todo o trabalho e as propostas de trabalhos futuros.

2 AGENTES RACIONAIS

2.1 Agentes Inteligentes

Tudo que é capaz de perceber seu ambiente por meio de sensores e agir através de atuadores é conhecido como agente (RUSSELL; NORVIG, 2004). Qualquer ser vivo pode ser considerado um agente se possuir, por exemplo, olhos, ouvidos ou quaisquer órgãos sensores; mãos, pernas, braços ou outras partes que sirvam como atuadores. De forma semelhante, um agente robótico pode ter câmeras, microfones, entre outros dispositivos sensoriais; braços e pernas mecânicos e diversos atuadores. A Figura 2.1 mostra uma abstração do conceito de agente.

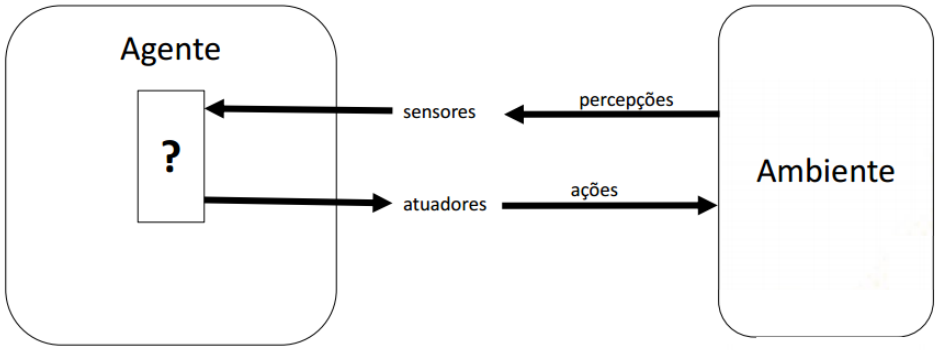

Figura 2.1 Estrutura básica de um agente.

Os sensores de um agente captam o estado do ambiente em um dado momento, esse estado é uma entrada do sensor mapeada na forma de percepção. Percepção é a informação sobre uma ou mais características do ambiente no momento, por exemplo, se o agente consegue, através de algum de seus sensores, perceber níveis de luminosidade, então ele será capaz de obter a informação sobre se o ambiente está iluminado ou escuro. O histórico completo de tudo que o agente já percebeu é chamado de sequência de percepções e interfere diretamente na escolha de uma ação.

A função de agente descreve matematicamente o comportamento do agente, mapeando as sequências de percepções em uma ação, esta função é implementada por um programa de agente. Distinguir a função de agente, do programa de agente é importante pois, a função é uma descrição matemática abstrata do comportamento do agente, enquanto o programa é a implementação concreta da função e está relacionado à arquitetura do agente (RUSSELL; NORVIG, 2004).

Um exemplo simples que pode ilustrar bem esses conceitos é o mundo do agente aspirador de pó (RUSSELL; NORVIG, 2004). A simplicidade desse mundo permite descrever tudo o que acontece. O mundo do aspirador de pó tem apenas dois locais, o quadro A e o quadro B. O agente aspirador de pó possui rodas e um mecanismo se sucção de sujeira como atuadores e seus sensores são câmeras. Ele pode perceber em que quadro ele se encontra e se o quadro possui sujeira ou não. As ações que ele pode optar em executar são mover-se para a direita, mover-se para a esquerda, aspirar ou não fazer nada. A princípio, pode-se pensar numa função de agente bem elementar que consiste em aspirar caso quadro atual estiver sujo, caso contrário mover-se para o quadro ao lado (RUSSELL; NORVIG, 2004). O agente aspirador de pó é ilustrado na Figura 2.2, na Figura 2.3 é apresentada uma tabulação parcial de sua função de agente.

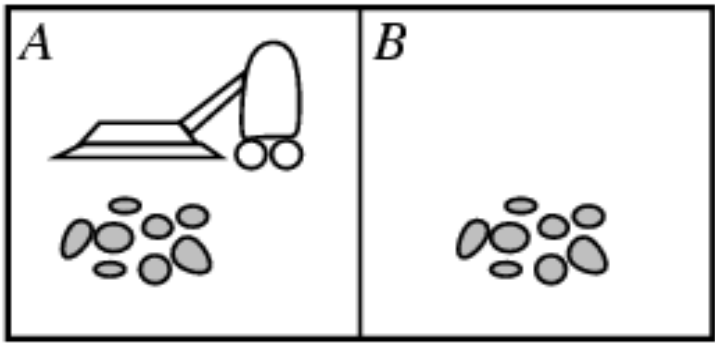

Figura 2.1 O mundo do agente aspirador de pó.

Sequência de Percepções	**Ação**
[A, Limpo]	Direita
[A, Sujo]	Aspirar
[B, Limpo]	Esquerda
[B, Sujo]	Aspirar
[A, Limpo], [A, Limpo]	Direita
[A, Limpo], [A, Sujo]	Aspirar
...	...
[A, Limpo], [A, Limpo], [A, Limpo]	Direita
[A, Limpo], [A, Limpo], [A, Sujo]	Aspirar
...	...

Figura 2.2 Tabela parcial da função de agente correspondente ao agente aspirador de pó.

É dito inteligente um agente que age com racionalidade, que faz tudo certo, em termos técnicos toda entrada à função de agente corresponde à ação esperada (RUSSELL; NORVIG, 2004). A primeira dificuldade é definir o que é fazer tudo certo. Pode-se considerar inicialmente que fazer certo é escolher a ação que permita ao agente obter maior sucesso. Decorre disso a necessidade de um método para medir o sucesso, esse método junto dos sensores e atuadores e da descrição do ambiente, fornece uma especificação completa da tarefa que o agente deve empreender.

A medida de desempenho é o critério para se medir o sucesso do comportamento do agente (RUSSELL; NORVIG, 2004). Ao ser inserido no ambiente, o agente gera uma sequência de ações de acordo com as percepções que recebe. Essas sequências de ações faz o ambiente passar por uma sequência de estados, se essa sequência é desejável, o agente funcionou bem. Considere por exemplo que a medida de desempenho definida para o agente aspirador de pó seja a quantidade de sujeira aspirada. Um agente inteligente pode maximizar essa medida de desempenho aspirando sujeira, despejando-a no chão logo em seguida e aspirando-a novamente, repetindo o processo diversas vezes.

Alternativamente, uma medida de desempenho mais adequada recompensaria o agente por deixar o chão limpo, para cada quadro limpo num período de tempo. Portanto, é recomendado construir medidas de desempenho de acordo com o resultado desejado no ambiente, em vez de seguir o comportamento esperado do agente, pois como foi demonstrado o agente pode agir de forma a maximizar seu desempenho, sem atingir o resultado esperado.

Selecionar uma medida de desempenho não é uma tarefa fácil. A mesma limpeza da medida de desempenho do parágrafo anterior pode ser alcançada por dois agentes diferentes, dos quais um faz um trabalho ordinário o tempo todo, enquanto o outro trabalha freneticamente, mas faz longas paradas. Introduzir penalidades na medida de desempenho pode ajudar a tornar o agente mais eficiente, por exemplo, ele poderia ser penalizado pela energia consumida e pelo barulho gerado.

O segundo problema é definir o que é racionalidade, porém quatro fatores ajudam a superar essa barreira, são eles: a medida de desempenho que define o critério de sucesso, o conhecimento anterior que o agente tem do ambiente, as ações que o agente pode executar e a sequência de percepções do agente até o momento (RUSSELL; NORVIG, 2004). Os conceitos de agente e de racionalidade conduzem a definição de um agente inteligente:

Para cada sequência de percepções possível, um agente racional deve selecionar uma ação que maximize sua medida de desempenho, dada a evidência fornecida pela

sequência de percepções e por qualquer conhecimento interno do agente (RUSSELL; NORVIG, 2004).

A racionalidade determina que um agente inteligente possua outras características importantes, são essas características que o permitem se comportar de maneira bem-sucedida. Ele deve ser capaz de coletar informações com a finalidade de modificar percepções futuras, possuir a habilidade de aprender com o propósito de modificar e ampliar seu conhecimento; e por fim, um agente inteligente deve ser autônomo, ou seja, ele aprende para compensar seu conhecimento anterior parcial ou incorreto e age com base nesse novo conhecimento adquirido, não apenas naquele conhecimento que foi fornecido pelo projetista.

2.2 Ambientes de Tarefa

Ambientes de tarefa podem ser considerados como os problemas para os quais os agentes inteligentes são as soluções (RUSSELL; NORVIG, 2004). Por conta da relação de dependência entre a função de agente, o programa de agente e a arquitetura do agente – que deve ser adequada para o ambiente em que o agente atuará, é preciso antes de qualquer coisa em um projeto de agente, especificar o ambiente de tarefas.

A especificação do ambiente de tarefas é feita através do agrupamento das definições da medida de desempenho, do ambiente, dos atuadores de dos sensores, este agrupamento recebe o acrônimo de PEAS (do inglês *Performance, Environment, Actuators, Sensors* – desempenho, ambiente, atuadores, sensores) (RUSSELL; NORVIG, 2004).

Primeiro define-se qual deve ser a medida de desempenho do agente, em termos de quais serão os critérios usados para medir o sucesso do agente; na sequência descreve-se o ambiente onde o agente será inserido, considerando as dificuldades que podem aparecer e características gerais do ambiente; depois são definidos os seus atuadores e sensores que devem ser adequados ao ambiente já descrito.

O mundo do agente aspirador de pó pode ser descrito facilmente já que é bem simples, servindo como um exemplo de PEAS elementar. A medida de desempenho adotada recompensa o agente por deixar o chão limpo e penaliza-o por consumir energia ou por provocar ruídos. O ambiente consiste em dois quadros. Os atuadores do agente são suas rodas e o seu mecanismo de sucção de sujeira, seus sensores são câmeras.

2.2.1 Propriedades dos Ambientes de Tarefas

Apesar da variedade imensa de ambientes de tarefas existente, estes podem ser divididos em categorias por suas propriedades. Essas propriedades determinam o projeto adequado de agentes e a aplicabilidade de cada uma das principais técnicas de implementação de agentes (RUSSELL; NORVIG, 2004). A seguir é fornecida uma lista de tipos de PEAS e suas definições:

- Completamente observável x Parcialmente observável: se os sensores do agente são capazes de acessar o estado completo do ambiente em cada momento, o ambiente de tarefa é completamente observável. É definido como estado completo do ambiente o conjunto de todos os aspectos relevantes para a escolha da ação, sendo que a relevância destes aspectos depende da medida de desempenho. Um ambiente é parcialmente observável quando há interferência, os sensores são imprecisos, ou quando o estado simplesmente não está completamente presente nos sensores (RUSSELL; NORVIG, 2004).
- Determinístico x Estocástico: um ambiente é determinístico se o próximo estado é obrigatoriamente determinado pelo estado atual e pela ação do agente; é estocástico caso contrário. Quando um ambiente é parcialmente observável ele poderá parecer estocástico, principalmente quando este é complexo, dificultando o controle de todos os aspectos não-observados. Portanto, geralmente é melhor distinguir um ambiente determinístico ou estocástico do ponto de vista do agente. Se o ambiente é determinístico, a não ser pela ação de outros agentes, então o ambiente é estratégico (RUSSELL; NORVIG, 2004).
- Estático x Dinâmico: ambientes estáticos não mudam enquanto o agente decide. Ambientes estáticos são particularmente fáceis de lidar, pois o agente não precisa observar o ambiente enquanto decide sobre a execução de uma ação, tão pouco se preocupa com a passagem do tempo. Já o ambiente dinâmico é bem complicado, pois exige do agente atenção contínua a suas mudanças. O ambiente é semidinâmico se ele não muda com a passagem do tempo, mas o nível de desempenho do agente se altera (RUSSELL; NORVIG, 2004).

- Episódico x Sequencial: quando um ambiente é episódico, a experiência do agente pode ser dividida em episódios, que são formados pela percepção e pela execução de uma única ação. É indispensável que o próximo episódio seja independente da ação executada no episódio anterior. Nesse tipo de ambiente a escolha da ação em cada episódio depende apenas do próprio episódio. Ambientes episódicos são bem mais simples que ambientes sequências porque o agente não precisa fazer previsões. A escolha da ação em cada episódio só depende do próprio episódio (RUSSELL; NORVIG, 2004).
- Discreto x Contínuo: para distinguir um ambiente discreto de um ambiente contínuo, deve-se considera aspectos a respeito do estado do ambiente, do modo como o tempo é tratado e das percepções e ações do agente. Um ambiente discreto possui número limitado e claramente definido de estados distintos, percepções e ações; e o tempo passa por intervalos discretos, no jogo de xadrez sem relógio, por exemplo, o tempo é representado discretamente por cada uma das rodadas (RUSSELL; NORVIG, 2004).
- Agente único x Multiagente: um ambiente é dito de agente único quando existe apenas um agente operando no ambiente; e multiagente caso contrário. Essa é uma definição bem simples, contudo ela tem um grande problema; ignora o que deve ser considerado um agente no ambiente. Um agente A pode assumir que um objeto B é um agente ou tratá-lo apenas como um objeto de comportamento estocástico presente no ambiente. O objeto B será definido corretamente como um agente se seu comportamento for descrito por uma medida de desempenho, a qual tenta maximizar, cujo valor depende do comportamento de A. Existem ainda dois subtipos de ambientes multiagente, o ambiente multiagente cooperativo e o ambiente multiagente competitivo. No ambiente multiagente competitivo os agentes tentam maximizar sua medida de desempenho, o que provoca a minimização da medida de desempenho dos demais agentes. Já nos ambientes multiagentes cooperativos, ao buscar a maximização de sua medida de desempenho cada agente individualmente está colaborando para a maximização da medida de desempenho de todos os agentes (RUSSELL; NORVIG, 2004).

2.3 Tipos de Agente

Ao definir o que é uma função de agente a primeira coisa que se pode imaginar é uma tabela onde para cada entrada de sequência de percepções há uma ação correspondente. O problema é que essa tabela é infinita, o que torna impraticável um agente dirigido por tabela. O propósito da IA é descobrir como criar programas capazes de produzir um comportamento racional a partir de uma pequena quantidade de código, e não a partir de uma tabela enorme com todas as entradas possíveis devidamente preenchidas (RUSSELL; NORVIG, 2004).

Existem quatro tipos básicos de agentes que atendem aos propósitos da IA, são eles os agentes reativos simples, agentes reativos baseados em modelo, agentes reativos baseados em objetivos, os agentes baseados na utilidade e os agentes com aprendizagem (RUSSELL; NORVIG, 2004).

2.3.1 Agentes Reativos Simples

Este é o tipo mais simples de agente, ele seleciona a ação com base apenas na percepção atual, desconsiderando o restante do histórico de percepções. O agente reativo simples utiliza de uma conexão entre uma entrada e uma condição que é chamada de regra condição-ação (RUSSELL; NORVIG, 2004).

A simplicidade dos agentes reativos simples tem um preço; eles possuem inteligência muito limitada, eles só funcionam bem em ambientes completamente observáveis, estão sujeitos a entrar em laços de repetição infinitos e uma impossibilidade de observação pode causar dificuldades. A Figura 2.4 apresenta um modelo básico de agente reativo simples.

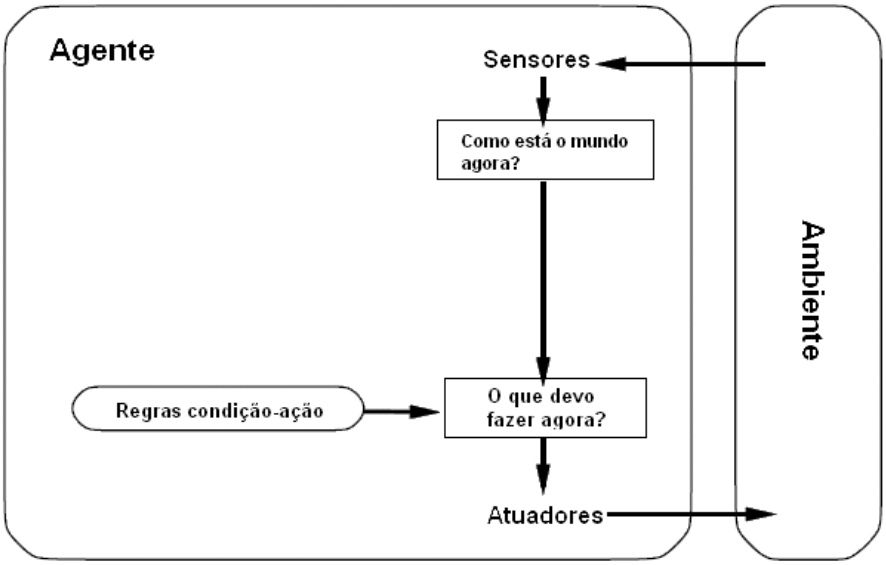

Figura 2.3 Estrutura de um agente reativo simples

2.3.2 Agentes Reativos Baseados em Modelo

Este tipo de agente lida muito bem com ambientes parcialmente observáveis, ele mantém um estado interno que depende do histórico de percepções e com isso é capaz de representar alguns aspectos não observados do estado atual (RUSSELL; NORVIG, 2004).

É bastante claro que estas informações internas de estado devem ser atualizadas à medida que o tempo passa, então é necessário codificar no programa de agente, dois tipos de conhecimento. O agente deve possuir algumas informações de como o mundo evolui independentemente dele, ele também precisa de informações de como as próprias ações afetam o mundo. Esse conhecimento do mundo e de como funciona é chamado de modelo (RUSSELL; NORVIG, 2004).

A Figura 2.4 ilustra a estrutura básica de um agente baseado em modelos.

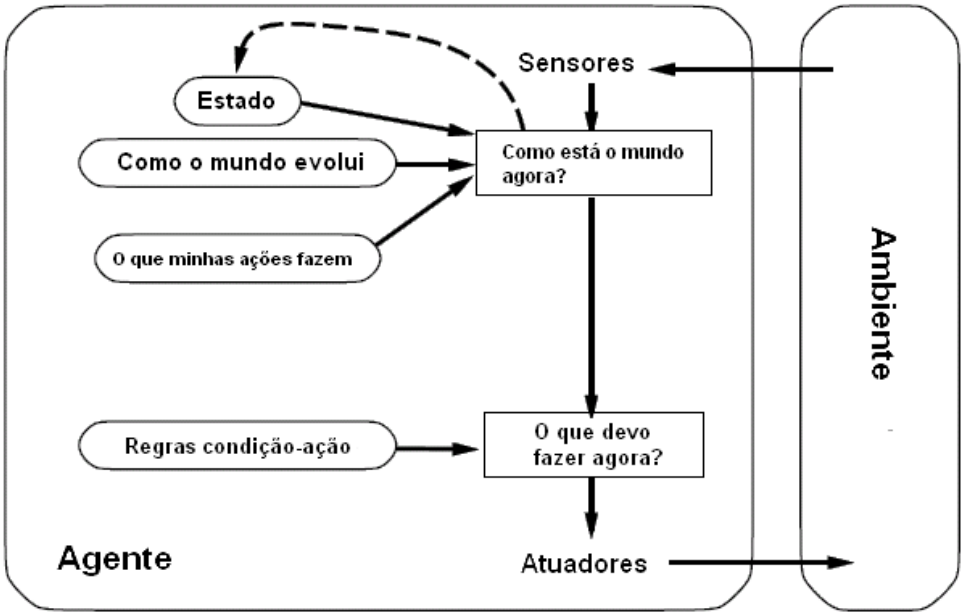

Figura 2.4 Estrutura de um agente reativo baseado em modelos.

2.3.3 Agentes reativos baseados em objetivos

Em certos casos conhecer o estado atual do ambiente não é o suficiente para que o agente decida sobre qual ação executar, ele deve possuir informações sobre objetivos que representam situações desejáveis. O programa de agente pode combinar os objetivos com informações sobre o resultado de ações possíveis, com o propósito de escolher ações que alcancem estes objetivos (RUSSELL; NORVIG, 2004).

A escolha das ações pode ser complexa, quando o agente deve considerar longas sequências de ações até encontrar uma forma de atingir seu objetivo. Quando essa situação acontece, o agente faz uso de métodos conhecidos como busca e planejamento, que se dedicam a encontrar sequências de ações que alcançam os objetivos do agente. Por vezes, a escolha das ações pode ser também bem simples, quando o objetivo é alcançado pela execução de uma única ação.

O agente baseado em objetivos tem a caraterística de ser bem flexível, pois o conhecimento que apoia suas decisões está representado explicitamente e pode ser modificado. Seu comportamento pode ser alterado com facilidade, uma vez que não serão necessárias mudanças em várias regras condição-ação como nos agentes reativos. A Figura 2.5 apresenta um modelo de agente baseado em objetivos.

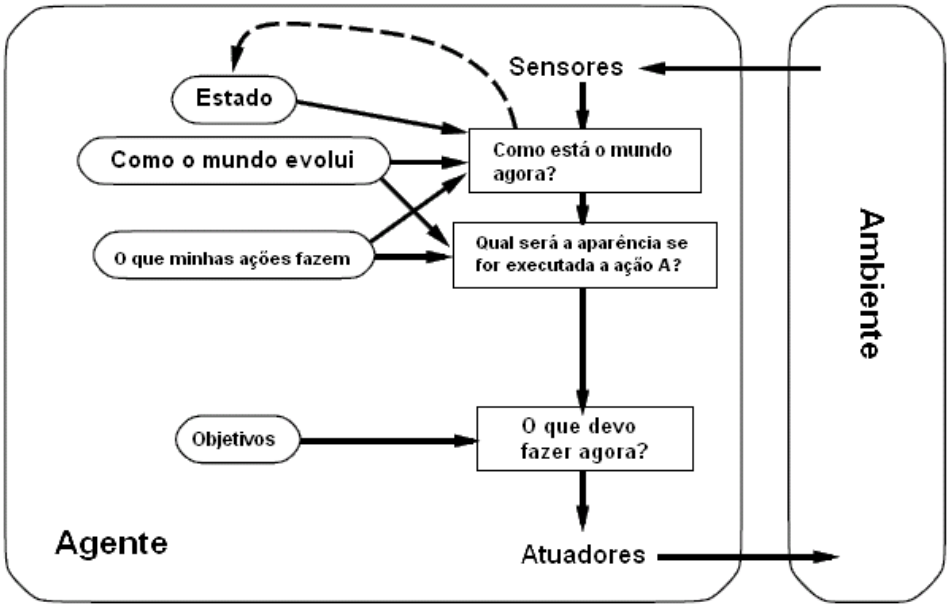

Figura 2.5 Estrutura de um agente baseado em objetivos.

2.3.4 Agentes baseados na utilidade

Apenas objetivos não bastam para se projetar agentes com comportamento de alta qualidade na maioria dos ambientes. Um dos problemas dos agentes baseados em objetivos é quando várias ações distintas levam ao mesmo objetivo, cada uma com suas vantagens e desvantagens, custos e benefícios. Objetivos simplesmente possibilitam a distinção binária entre "estados desejáveis" ou "indesejáveis", uma medida de desempenho geral, por sua vez, permite a comparação entre estados do mundo, com o grau exato de satisfação que proporcionariam ao agente, se fossem alcançados. Considerando que "desejável" não parece muito científico, é mais apropriado dizer que, se um estado do mundo for mais vantajoso que outro, ele é mais útil para o agente.

Os agentes baseados na utilidade possuem uma função de utilidade que associa um estado (ou uma sequência de estados) a um número real, que descreve o grau de satisfação do objetivo, ou seja, o quanto o agente foi eficiente. A especificação completa da função de utilidade permite decidir racionalmente em casos em que os objetivos são inadequados (RUSSELL; NORVIG, 2004).

Um deles é quando existem objetivos contraditórios e somente um deles pode ser alcançado, a função de utilidade determina o compromisso apropriado. Outra possibilidade é a de existirem vários objetivos e nenhum deles pode ser atingido certamente, a utilidade possibilita que a probabilidade de sucesso seja ponderada pela importância dos objetivos.

A Figura 2.6 apresenta um modelo de agente baseado na utilidade.

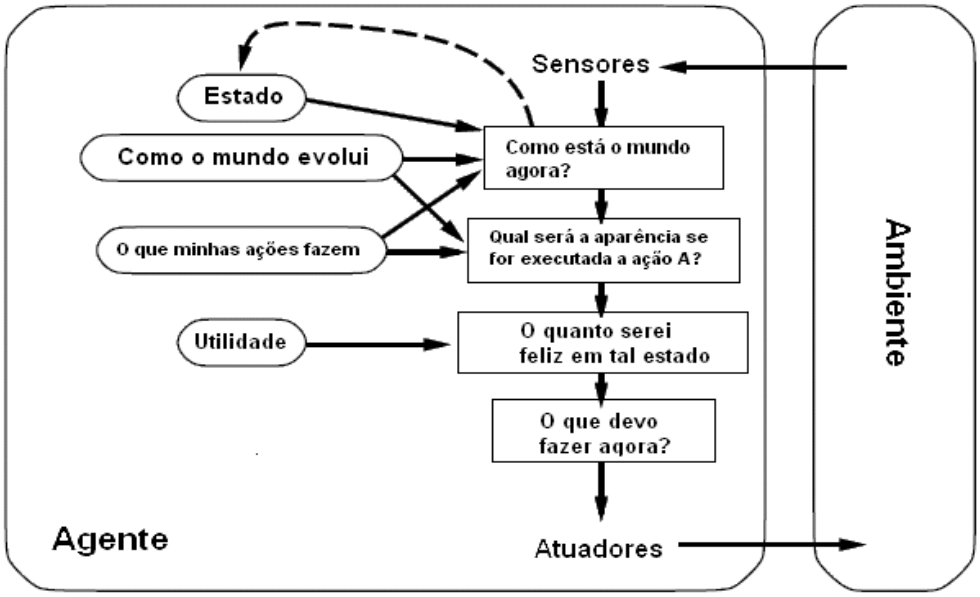

Figura 2.6 Estrutura de um agente baseado em utilidade.

2.3.5 Agentes com Aprendizagem

Aprendizagem é um método eficiente de programar agentes inteligentes, além de permitir ao agente operar em ambientes inicialmente desconhecidos e se tornar mais competente do que seu conhecimento inicial permitiria.

Um agente com aprendizado possui quatro componentes conceituais, chamados: elemento de aprendizagem, elemento de desempenho, crítico e gerador de problemas. O elemento de aprendizagem é responsável pelo aperfeiçoamento do agente, determinando modificações no elemento de desempenho, a partir de informações enviadas pelo crítico. O crítico por sua vez, avalia o desempenho do agente em relação a um padrão conhecido e fixo. O elemento de desempenho é o responsável pela escola das ações, que são direcionadas pelas informações que recebe. Por fim, o gerador de problemas deve indicar novas ações a serem experimentadas, de modo que possam ser descobertas maneiras de realizar as tarefas (RUSSELL; NORVIG, 2004).

A Figura 2.7 apresenta um modelo de agente com aprendiz

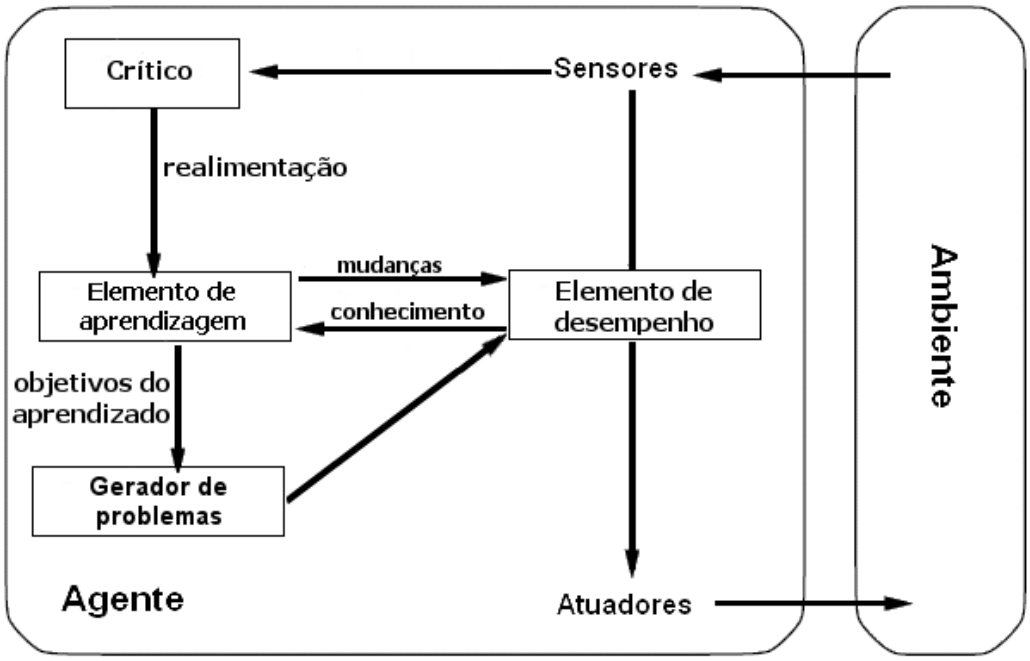

Figura 2.7 Agente com aprendizagem.

3 MUNDO DE WUMPUS E LÓGICA DE PREDICADOS

3.1 Mundo de Wumpus

O mundo de wumpus é uma caverna formada por um conjunto de salas ligadas entre si. Escondido em algum lugar na caverna está o wumpus, um monstro fedorento que devora qualquer um que entrar em sua sala. Algumas salas contêm poços sem fundo, nos quais cairá qualquer um que vagar por elas, exceto o wumpus que é muito grande para cair no poço. Dentro da caverna é possível encontrar ouro. O agente tem como tarefa entrar na caverna, explorá-la a fim de encontrar o ouro, não ser devorado pelo wumpus ou cair num poço e sair com vida. O agente possui um arco e uma flecha que pode utilizar para tentar matar o wumpus (RUSSELL; NORVIG, 2004). A Figura 3.1 representa uma das possíveis configurações do ambiente do mundo de wumpus.

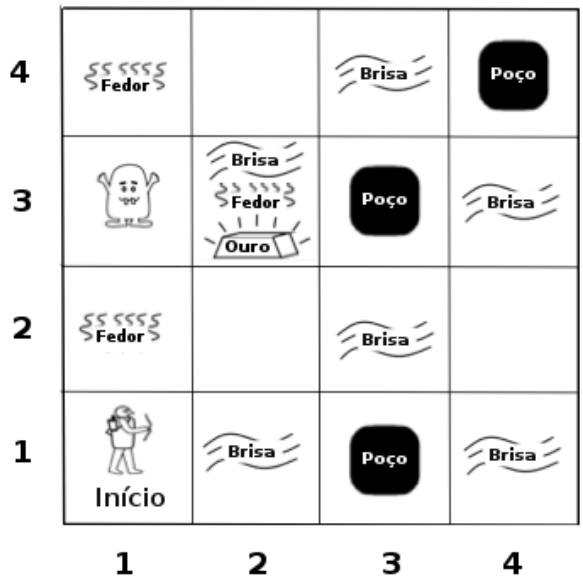

Figura 3.1 Exemplo clássico do mundo de Wumpus.

Este problema foi proposto inicialmente por Michael Genesereth. Abaixo segue uma descrição formal do PEAS do mundo de wumpus (RUSSELL; NORVIG, 2004):

- Medida de desempenho: +1000 por pegar o ouro, -1000 se cair no buraco ou for devorado pelo wumpus, -1 para cada ação executada e -10 pelo uso da flecha.
- Ambiente: uma matriz 4x4 de salas. Cada sala é representada por pares de

coordenadas na forma de [*linha, coluna*], por exemplo [3,2]. O agente sempre começa no quadro identificado como [1,1], voltado para a direita. As posições do ouro e do Wumpus são escolhidas ao acaso, devendo ser diferentes da sala inicial. Cada quadrado com exceção do inicial pode ser um poço, com probabilidade 0,2.

- Atuadores: o agente pode mover-se para frente, virar a esquerda ou direita. Ele morrerá se entrar numa sala contendo um poço ou o Wumpus vivo.

- Sensores: o agente tem cinco sensores, cada um retorna uma informação:

    - Na sala contendo o Wumpus e nas salas adjacentes não diagonais, o agente sentirá um fedor.

    - Nas salas adjacentes não diagonais a um poço ele sentirá uma brisa.

    - Na sala onde está o ouro o agente perceberá um brilho.

    - Quando o agente colidir com uma parede ele sentirá um impacto.

    - Quando o Wumpus é morto ele emite um grito que pode ser percebido pelo agente em qualquer lugar da caverna.

A dificuldade está no fato de que uma vez dentro da caverna, o agente é capaz de enxergar apenas a sala onde está, perceber o fedor nas salas ao redor do Wumpus e uma brisa nas salas adjacentes a uma sala que contêm um poço, porém não sabe a priori em qual sala exatamente se encontra o perigo.

As percepções serão dadas na forma de uma lista de cinco símbolos, por exemplo, se o agente percebe um fedor e uma brisa, não percebe brilho ou escutou um grito então a lista de percepções terá esta forma: [*fedor, brisa, nada, nada, nada*], se ele não percebe nada então a lista será [*nada, nada, nada, nada, nada*].

Segue abaixo uma descrição do PEAS do mundo de wumpus (RUSSELL; NORVIG, 2004):

- Parcialmente Observável: pois o agente não consegue perceber tudo o que existe na caverna no mesmo instante de tempo e a princípio, não tem conhecimento sobre nada a respeito de partes não observadas do ambiente.

- Determinístico: o estado seguinte é invariavelmente determinado pelo estado atual e pela ação que o agente resolve executar. Se a sala onde o agente se localiza não é uma sala que adjacente à outra que contem um poço, é impossível que o agente caia num poço. O estado futuro de estar morto por ter caído num poço, depende do estado anterior de estar numa sala vizinha à outra contendo um poço, depende ainda da ação de mover-se para a tal sala a perigosa.

- Não-Episódico: sempre que o agente está decidindo qual ação tomar, tal decisão depende diretamente da ação executada anteriormente. Por exemplo, sempre que o agente decide que é seguro entrar na sala onde estava o Wumpus, é porque antes ele decidiu tentar mata-lo e obteve sucesso.

- Estático: tudo o que existe no mundo de Wumpus não se altera enquanto o agente delibera. A morte do wumpus altera o ambiente, porém isso não torna o PEAS dinâmico, pois quando o agente decide disparar a flecha, a morte do wumpus acontece depois que ele decidiu atirar, se ele acertar. Outras decisões serão tomadas apenas após a flechar atingir o seu alvo, ou seja, ou o wumpus foi atingido e morreu, ou a flecha acetou uma parede, de qualquer forma a ação foi concluída.

- Discreto: o tempo no mundo de wumpus passa por intervalos discretos, como um jogo de xadrez sem relógio, cada instante de tempo é determinado pelo ciclo formado entre uma percepção e uma ação, semelhante a uma rodada do xadrez. Em cada momento existe um número limitado e bem definido de ações, estados e percepções, não existe a possibilidade, por exemplo, de o agente perceber um pouco de brisa, e em seguida perceber um pouco mais que antes.

Para exemplificar como funciona o mundo de wumpus, será demonstrado como um agente pode explorar o ambiente. Com base no ambiente da Figura 9(a), o agente sabe que ele está inicialmente em [1,1], voltado para a direita e conhece as regras do ambiente.

Neste momento a primeira percepção é [nada, nada, nada, nada, nada]. A Figura 3.2 apresenta o conhecimento do agente. Como o agente não percebeu brisa ou fedor em [1,1], ele pode deduzir que [1,2] e [2,1] não são salas perigosas. Essas salas estão marcadas com "*S*" para indicar que são seguras. O agente é cauteloso, ele se move apenas se souber que a sala está marcada com "*S*". Suponha que o agente decidiu mover-se para frente até [1,2], como mostra a Figura 9(b). Agora, o agente percebe brisa em [2,1], isso implica que deve

haver um poço na vizinhança. Apesar de a sala [1,1] ser vizinha de [2,1], tal poço não pode estar lá porque isso é contra as regras do ambiente, sendo assim, o poço pode estar em [2,2] ou [3,1] ou em ambos. A notação "*PP*" da Figura 3.2(a) indica um possível poço nessas salas. Existe apenas uma sala segura e não visitada é conhecida neste instante, consequentemente, o agente sendo prudente pode decidir retornar a [1,1] e depois avança para [1,2].

Em [1,2] a nova lista de percepções formada é [*fedor*, *nada*, *nada*, *nada*, *nada*], levando ao estado representado na Figura 3.3(b). O fato de o agente perceber fedor em [1,2] significa que o wumpus está por perto. Contudo, de acordo com as regras do jogo o wumpus não pode estar em [1,1], também não está em [2,2], pois o agente não detectou fedor em [2,1]. Sendo assim, o agente deduz que o wumpus está em [1,3]. Ainda nesse instante, o agente não percebe brisa em [1,2], o que elimina a possibilidade de existir um poço em [2,2]. Como o agente cogitou a existência de um poço em [2,2] e/ou [3,1], ao constatar que [2,2] não contem um poço, ele também é capaz de deduzir que o poço está em [3,1]. Esse tipo de inferência não é simples, pois combina o conhecimento que foi obtido em diferentes lugares em diferentes momentos, além de usar a falta de uma percepção para tomar decisões.

Após ter a certeza que não há um poço ou o wumpus em [2,2], então o agente sabe que esta sala é segura e que pode visita-la.

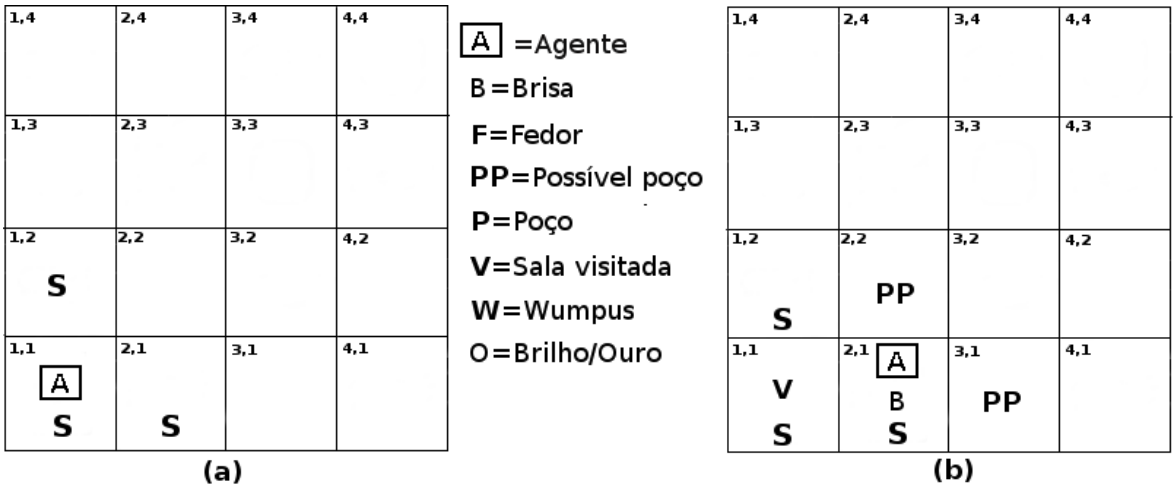

Figura 3.2 Explorando o ambiente.

Ao chegar em [2,2] a nova lista de percepções é [*nada*, *nada*, *nada*, *nada*, *nada*], então o agente deduz que todas as salas vizinhas são seguras e decide ir até [2,3]. Em [2,3] ele percebe [*fedor*, *brisa*, *brilho*, *nada*, *nada*], o que indica que há um poço e o wumpus na vizinhança, indica ainda que o ouro está na sala atual. Então o agente decide agarrar o ouro e com isso encerrar a busca, como mostra a Figura 3.3(b).

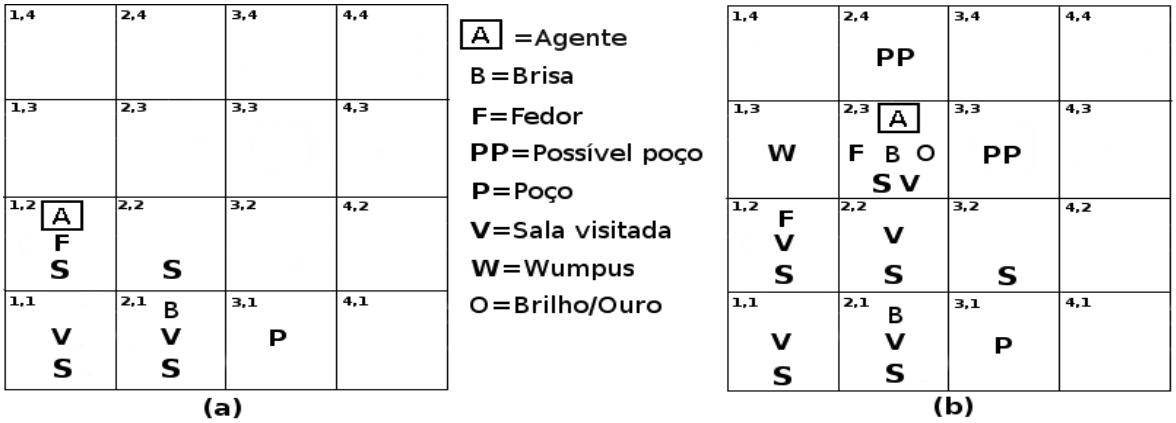

Figura 3.3 Como o agente explora e percebe o ambiente.

3.2 Agentes Baseados em Conhecimento

Um tipo de agente que figura como adequado para o problema do mundo de wumpus é o agente baseado em conhecimento, pois conhecimento e raciocínio permitem ao agente comportar-se de forma bem-sucedida, e também desempenham um papel fundamental ao lidar com ambientes *parcialmente observáveis*. Tal agente pode combinar o conhecimento geral com percepções correntes para deduzir aspectos ocultos do estado atual antes de selecionar ações. Seu componente principal é sua base de conhecimento que é um conjunto de sentenças. Uma sentença representa uma afirmativa sobre o mundo e é expressa em uma linguagem de representação de conhecimento. Deve haver um meio de adicionar sentenças à base de conhecimento e de consultar o que se conhece. Tanto a adição de sentenças quanto consultas que podem envolver inferência que é a derivação de novas sentenças a partir de sentenças antigas.

Agentes baseados em conhecimento obedecem ao princípio de que quando se consulta a base de conhecimento, a resposta deve seguir do que foi adicionado a ela. Um agente baseado em conhecimento recebe uma percepção e retorna uma ação como todos os outros. Ele mantem uma base de conhecimento que pode conter inicialmente algum conhecimento geral. Toda vez que o programa de agente é chamado, ele primeiro informa à

base de conhecimento sua percepção. Em seguida, pergunta à base de conhecimento que ação deve executar. Ao processar a pergunta, possivelmente será necessário desenvolver raciocínios sobre o estado atual do mundo, sobre os resultados de sequências de ações e etc. (RUSSELL; NORVIG, 2004).

3.3 Lógica de Predicados

Para programar um agente baseado em conhecimento, é preciso utilizar uma linguagem de representação de conhecimento que possua poder de representação suficiente para construir uma base de conhecimento, consulta-la e adicionar novas sentenças, possivelmente envolvendo inferência. A Lógica Proposicional não é totalmente adequada a esse proposito, pois, apesar de simples, o cálculo proposicional não consegue expressar fatos genéricos de modo conciso, como por exemplo, “se o agente estiver em qualquer local com o Wumpus à frente, não prosseguir em frente”. Seria interessante poder expressar tais fatos genéricos através de uma linguagem que pudesse fazer referência a objetos e a algumas de suas propriedades.

A Lógica de Predicados é uma linguagem de representação ideal neste caso, pois captura as relações entre objetos de um mesmo universo de discurso permitindo concluir particularizações de uma propriedade geral destes objetos, assim como derivar generalizações a partir de fatos que valem para um objeto arbitrário. Objetos podem ser qualquer componente do mundo com características individuais. Esse poder de expressão se deve ao fato de que ela usa um arsenal de símbolos mais sofisticados que a Lógica Proposicional, tornando-se muito mais expressiva que esta última. A Lógica de Predicados utiliza *modelos* para definir a verdade de uma sentença. Modelos são representações do mundo, sendo que os modelos da Lógica de Predicados são muito mais completos que os da Lógica Proposicional, pois possuem objetos que podem estar relacionados de diversas formas; contêm relações unárias, binárias, n-árias, propriedades e funções. É chamado de **domínio** o conjunto de todos os objetos de um modelo, os objetos também podem ser chamados de **elementos do domínio** (RUSSELL; NORVIG, 2004). Essas características ampliam o poder de representação. Por exemplo, caso o agente do mundo de wumpus estiver interessado em saber onde existem poços, ele pode usar a informação que decorre de uma sala em que ele percebe brisa, combinada com a definição do que é uma sala segura e todas as sentenças existentes na base de conhecimento até o momento para inferir onde realmente existe um poço.

3.3.1 Símbolos e Interpretações

Basicamente, a Lógica de Predicados é composta de símbolos que representam objetos, relações e funções. Devido a isso, os símbolos são de três tipos: **símbolos de constates** representando objetos; **símbolos de predicados** que representam propriedades ou relações entre objetos, e **símbolos de funções** que representam funções (RUSSELL; NORVIG, 2004). Os novos símbolos introduzidos são capazes de representar mais que simples sentenças lógicas que podem ser falsas ou verdadeiras; agora, se pode representar objetos, predicados e funções. Objetos são representações de coisas particulares do ambiente, como por exemplo, *wumpus*, *agente*, *ouro* e etc.. Predicados são propriedades de objetos ou relações entre eles. São como se fosse uma função em programação: eles recebem argumentos e têm uma resposta booleana, por exemplo, *poço([1,2])* será verdadeiro se existir um poço na sala [1,2] no modelo em questão. Funções são parecidas com os predicados, porém, possuem apenas uma resposta e ela é sempre um elemento do domínio. Por exemplo, a função *agenteEm(X)* substitui variável "*X*" pela sala onde o agente se encontra no momento. Variáveis têm a mesma sintaxe das constantes, com a diferença de que as variáveis são usadas para se fazer consultas. No exemplo anterior, "*X*" assume o nome do objeto, no caso a sala onde o agente está. A **aridade** de um símbolo de predicado ou de função, diz respeito à quantidade fixa de argumentos que a função recebe (RUSSELL; NORVIG, 2004). Por exemplo, o predicado *adjacente(X,Y)* tem aridade igual a dois, já a função *agenteEm(X)* tem aridade igual a um.

A fim de determinar a verdade, sentenças são relacionadas a modelos, sendo assim, é necessária uma **interpretação** que especifique com exatidão quais objetos, relações e funções são referenciados pelos símbolos envolvidos (RUSSELL; NORVIG, 2004)**.** A seguir, são demonstradas possíveis interpretações do mundo de wumpus:

- *wumpusEm* se refere ao monstro, *adjacente* se refere à relação de adjacência ente salas,
- *agenteEm* se refere a função que retorna o local onde o agente está,
- *fedor*, *brisa*, *brilho*, *impacto* e *grito* são sentenças interpretadas em termos de o que o agente percebe no momento,
- *virar(direita)*, *pegar*, *atirar* e *mover* são sentenças interpretadas como ações.

A verdade de qualquer sentença é estabelecida por um modelo e uma interpretação para os símbolos desta. Portanto, consequência lógica, validade e etc. são definidas em termos de todos os modelos possíveis e todas as interpretações possíveis (RUSSELL; NORVIG, 2004).

3.3.2 Termos

Em Lógica de Predicados termos são definidos de acordo com as seguintes regras (PITANGUI, C; ZAVERUCHA, G, 2013):

1. Qualquer constante é um termo.
2. Variáveis são termos.
3. Toda expressão *f (t1,..., tn*) de $n \geq 1$ argumentos (onde cada argumento *ti* é um termo e *f* é um símbolo de função de aridade *n*) é um termo.

Não obstante, os símbolos de constantes sejam termos, nem sempre é viável ou conveniente utilizar um termo distinto para cada objeto. Por exemplo, podemos usar uma sentença do tipo *rodaBicilceta(Dianteira)* ao invés de dar um nome à roda. Termos complexos são formados por um símbolo de função seguido por uma lista de argumentos

3.3.3 Sentenças Atômicas

Sentenças atômicas enunciam fatos e são formadas pela união de objetos e relações. Sua forma se dá a partir de um símbolo de predicado seguido por uma lista de termos entre parênteses. Por exemplo:

Poço([4,1])

Isso afirma que existe um poço em [4,1]. Sentenças atômicas também podem conter termos complexos como argumentos. Como por exemplo:

Neto(X,(Pai(Y,X),PAI(Z,Y)))

3.3.4 Sentenças Complexas

Sentenças complexas são obtidas ao usar conectivos lógicos em conjunto com sentenças atômicas (RUSSELL, NORVIG, 2004). O resultado são sentenças parecidas com as da lógica proposicional. Exemplos:

$\neg Brisa([1,2])$

$\neg Brisa([1,2]) \wedge \neg Fedor([1,2])$

3.3.5 Quantificadores

Quantificadores são usados para expressar usados para expressar regras gerais e propriedades de coleções de objetos. A falta desse mecanismo é o principal problema da lógica proposicional; regras como "Todas as salas adjacentes ao wumpus são fedorentas" só poderiam ser expressas enumerando essas salas adjacentes uma a uma pelo nome. A Lógica de Predicados possui dois quantificadores chamados de **quantificador universal** e **quantificador existencial**.

3.3.6 Quantificador Universal

O quantificador universal cujo símbolo é $\forall$, é usado para expressar regras gerais em logica de predicados. Geralmente, ele significa "Para todo(s)..." ou "Qualquer que seja...".

Combinando este símbolo com variáveis e os demais elementos da lógica de predicado, é possível formar sentenças como:

$\forall x\ Moto(x) \Rightarrow duasRodas(x)$, significado: "Para todo e qualquer x, se x é uma moto, então x possui duas rodas".

$\forall x\ Rato(x) \Rightarrow Roedor(x)$, significado: "Para todo e qualquer x, se x é um rato, então x é um roedor".

3.3.7 Quantificador Existencial

O quantificador existencial, cujo símbolo é $\exists$, serve para fazer declarações a respeito de algum objeto sem nomeá-lo. Ele expressa se existe ou não ao menos um elemento para o qual a declaração feita é verdadeira. Com este quantificador é possível formar sentenças do tipo:

$\exists x$ *Motorista(x)* $\wedge$ *Estudante(x)*, significado: “Existe um motorista que é estudante”.

$\exists x$ *Roedor(x)* $\wedge \neg$ *Rato(x)*, significado: “Existe um roedor que não é um rato”.

3.4 Buscando a Solução Através da Lógica de Predicados

Enquanto linguagem de representação de conhecimento, a Lógica de Predicados captura de forma natural o que se deseja expressar. A seguir é apresentado como a Lógica de Predicados pode ajudar na tarefa de representar o conhecimento do agente do mundo de wumpus (RUSSELL, NORVIG, 2004).

Em Lógica de Predicados, uma sentença correspondente a uma percepção do agente do mundo de wumpus, deve conter além da lista de percepções e uma forma de controlar o passar do tempo, caso contrário, o agente ficará confuso sobre o momento em que percebeu cada elemento. Por exemplo:

Percepção([Fedor, Brisa, Brilho, Nada, Nada],Tempo).

Neste exemplo, *Percepção* é um predicado de *aridade* igual a dois, e cada elemento do vetor pode ser representado por um termo lógico constante: a variável *Fedor*, por exemplo, o termo poderia ser *fedor* caso ele perceba fedor, ou *nada* caso contrário.

As ações do agente podem ser representadas por termos ou sentenças atômicas, como no exemplo a seguir:

Virar(direita), Virar(esquerda), Avançar, Atirar, Agarrar, Soltar

A fim de saber qual ação deve ser executada; o programa de agente deve formular uma consulta à base de conhecimento semelhante a que se segue:

$\exists x$ *Ação(x,5).*

A consulta anterior possui dois argumentos, o primeiro argumento é a variável *x* que receberá a ação que deverá ser executada, de acordo com a base de conhecimento, o segundo argumento representa o instante (tempo) que no exemplo é igual a cinco. Embora a decisão sobre qual ação executar dependa, entre outros fatores, das percepções atuais, a consulta anterior considera que tanto a percepções quanto os demais fatores necessários, já fazem parte da base de conhecimento, por isso não é preciso passa-los como argumento.

Os dados obtidos da percepção do agente demonstram alguns fatos sobre o estado atual do ambiente. Por exemplo:

$\forall t, f, b, o, m, c$ *Percepção*$([f, b, o, m, c], t) \Rightarrow$ *Brisa*(t).

A sentença anterior indica um estado do ambiente que é o fato do agente perceber brisa na sala atual, ou seja, a sala que se encontra no instante *t*,

$\forall t, f, b, o, m, c$ *Percepção*$([f, b, o, m, c], t) \Rightarrow$ *Brilho*(t).

Essa sentença representa o estado do ambiente no qual o agente percebe o brilho do ouro na sala em que ele está no instante *t*.

É possível ainda, utilizar sentenças de implicação quantificadas, simulando um comportamento reativo simples. Por exemplo:

$\forall t$ *Brilho* $\Rightarrow$ *Ação*(*Pegar*, t).

A sentença anterior é semelhante a uma regra de condição-ação do agente reativo simples, sua condição é a percepção do brilho do ouro e sua reação é a ação de pegar o ouro.

Após representar percepções e ações, é hora de representar o ambiente. A melhor maneira de representar as salas é utilizar termos complexos em que linha e coluna são números inteiros, por exemplo, o termo em forma de lista *[1,2]* indica a sala que se encontra na linha 1 e coluna 2. A relação de adjacência entre as salas pode ser representada como:

$\forall x, y, a, b$ *Adjacente*$([x, y], [a, b]) \Leftrightarrow [a, b] \in \{[x+1, y], [x-1, y], [x, y+1], [x, y-1]\}$.

A regra anterior determina a forma como se dá a relação de adjacência entre salas, afirmando que são adjacentes a uma sala qualquer, as salas que se localizam uma linha imediatamente acima ou abaixo; ou uma coluna imediatamente anterior ou posterior à sala atual. Isso exclui as salas diagonais, como as regras do ambiente impõem.

O Wumpus e os poços podem ser representados perfeitamente por predicados unários, uma vez que o wumpus não sai do lugar e que não faz sentido fazer distinção entre poços. Exemplo:

Wumpus(x). A variável *x* guarda a posição onde se encontra o wumpus.

Poço(p). Ao consultar este predicado passando como argumento uma sala ele retorna verdadeiro se lá existe um poço, caso contrário retorna falso.

Dada à posição atual do agente, ele pode deduzir características da sala a partir de sua percepção corrente. Por exemplo, se o agente estiver em uma sala e perceber brisa, então essa sala é arejada:

$\forall s, Em(Agente, s, t) \wedge Brisa(t) \Rightarrow Arejado(s)$. A variável *t* serve para indicar o instante.

Para o agente, saber quais são as salas arejadas ou fedorentas e quais não são, é muito importante, pois a partir desse conhecimento ele pode deduzir onde estão os poços e o wumpus. Há regras que podem ajudar em tais deduções, elas se denominam Regras de Diagnóstico.

3.4.1 Regras de Diagnóstico

Regras de diagnóstico são originadas de efeitos observados de causas ocultas (RUSSELL; NORVIG, 2004). Com a finalidade de encontrar poços, regras de diagnóstico determinam que, se uma sala é arejada, alguma sala adjacente contém um poço:

$\forall s\ Arejado(s) \Rightarrow \exists r\ Adjacente(r, s) \wedge Poço(r)$.

Afirmam ainda que, se uma sala não é arejada, nenhuma sala adjacente contém um poço:

$\forall s\ \neg Arejado(s) \Rightarrow \neg \exists r\ Adjacente(r, s) \wedge Poço(r)$.

Ao se combinar as duas, forma-se a sentença bicondicional:

$\forall s\ Arejado(s) \Leftrightarrow \exists r\ Adjacente(r, s) \wedge Poço(r)$.

3.4.2 Regras Causais

As regras causais refletem o fato de que certas propriedades ocultas do mundo podem provocar a geração de certas percepções (RUSSELL; NORVIG, 2004). Por exemplo, um poço é a **causa** de salas adjacentes serem arejadas:

$\forall r\ Poço(r) \Rightarrow [\forall r\ Adjacente(r, s) \Rightarrow Arejado(s)]$

E, se todas as salas adjacentes a uma dada sala não contiverem poços, a sala não será arejada:

$\forall s\ [\forall r\ Adjacente(r, s) \Rightarrow \neg Poço(r)\ ] \Rightarrow \neg Arejado(s)]$

Dessa forma, fica muito claro o poder de representação da Lógica de Predicados, e como ela pode representar o mundo de wumpus de uma forma muito mais concisa que a descrição em linguagem natural.

4 FERRAMENTAS UTILIZADAS E PROGRAMAÇÃO

4.1 Software e Hardware

Este capítulo apresenta as ferramentas que foram utilizadas para criar o agente por completo. A arquitetura utilizada foi o kit LEGO Mindstorms. Entretanto, o kit não possui o poder de processamento necessário para lidar com programação lógica, dessa forma, com o propósito de contornar este obstáculo, tornou-se necessário adicionar um computador à arquitetura. O computador comunica-se com o kit através de Bluetooth.

Para definir a base de conhecimento do agente foi utilizada a linguagem de programação Prolog, e o software escolhido para esta tarefa foi o SWI-Prolog. Nesse ponto surgiu outro problema; programas feitos em Prolog não são capazes de se comunicar diretamente com o kit LEGO. A alternativa foi criar um programa principal que fosse capaz de se comunicar com ambos, kit e a base de conhecimento em Prolog. A linguagem escolhida para construir este programa foi C++, pois existe uma interface para ele implementada no SWI-Prolog.

Vários fatores foram considerados antes de se fazer qualquer escolha, maiores detalhes sobre a escolha de cada ferramenta serão apresentados em seus respectivos tópicos.

4.2 SWI-Prolog

Uma vez definida a base de conhecimento, é hora de programa-la para que possa ser utilizada junto à arquitetura do agente. Porém, antes é preciso determinar qual a melhor forma de se criar esse programa.

O Paradigma de Programação Lógico é baseado na utilização de sentenças lógicas, utilizando-se da lógica simbólica como linguagem para programação. Um dos pontos importantes são os objetos e seus relacionamentos, onde se podem declarar vários fatos sobre estes, definir regras e questionamentos onde as respostas podem ser observadas através das regras e fatos. O Paradigma Lógico é bastante eficiente para exemplificar modelos de estruturas de dados e códigos, simplificando ideias complexas, onde um problema pode ser mostrado por meio de relações sobre um determinado conjunto de objetos.

A Programação Lógica é bastante utilizada no campo da inteligência artificial, onde se busca implantar a maneira da lógica matemática à programação de computadores. O

paradigma lógico elabora deduções imediatas com base em uma lista de premissas.

Prolog é a principal linguagem de programação lógica. Esta ganhou espaço no mercado com os sistemas baseados em técnicas de representação da Inteligência Artificial e do conhecimento.

Em Prolog a aridade de um predicado ou regra segue o mesmo modelo da Lógica de Predicados, ou seja, depende do seu número de argumentos. O protótipo de qualquer predicado ou função demonstra sua aridade, por exemplo, a função a*genteEm/1* tem aridade igual a 1, o predicado *agenteVivo/0* tem aridade zero, portanto não recebe argumentos.

O SWI-Prolog oferece um ambiente Prolog livre e abrangente. Desde o seu início em 1987, o desenvolvimento SWI-Prolog tem sido impulsionado pelas necessidades de aplicações do mundo real. SWI-Prolog é amplamente utilizado em pesquisa e educação, bem como aplicações comerciais.

A maioria das implementações da linguagem Prolog são projetadas para servir a um conjunto limitado de casos de uso. SWI-Prolog se posiciona principalmente como um ambiente de programação mais geral, e em certos casos atua como uma ligação entre componentes de uma aplicação. Ao mesmo tempo, SWI-Prolog visa proporcionar um ambiente de prototipagem rápida produtiva. A sua orientação para a programação mais geral é apoiada por escalabilidade, velocidade de compilador, a estruturação do programa (módulos), suporte para *multithreading* para acomodar servidores, suporte a Unicode e interfaces para um grande número de formatos de documentos, protocolos e linguagens de programação. A prototipagem é facilitada por boas ferramentas de desenvolvimento, tanto para uso de linha de comando como para uso com ferramentas de desenvolvimento gráficas. O carregamento sobre demanda de predicados da biblioteca e a facilidade de compilação e ligação evita a necessidade de utilização de declarações e reduz a digitação.

SWI-Prolog é tradicionalmente forte na educação, não somente porque é livre e portátil, mas também por causa de sua compatibilidade com livros didáticos e seu ambiente fácil de usar. Porém, isso não implica que o sistema não pode ser utilizado em outros cenários. É utilizado como uma linguagem embutida onde serve como um pequeno subsistema em um aplicativo maior. É também usado como uma base de dados dedutiva.

Há uma série de razões pelas quais ele pode ser a melhor escolha a um sistema comercial, ou a outro sistema livre:

- O desempenho é a sua primeira preocupação:

  Vários sistemas livres e comerciais têm um melhor desempenho. Porém, os padrões Prolog de referência desconsideram muitos fatores que muitas vezes são críticos para o desempenho de grandes aplicações. SWI-Prolog não é bom em chamadas rápidas de predicados simples e seleções "se-então-senão" com base em testes internos simples, mas é rápido com código dinâmico e predicados que contêm um grande número de cláusulas. Muitos dos predicados internos de SWI-Prolog são escritos em C e tem excelente desempenho.

- Ambiente agradável:

  SWI-Prolog fornece um bom ambiente de linha de comando, incluindo autocompletar, histórico e um rastreador que opera em acionamento de teclas individuais. O sistema recompila automaticamente partes modificadas do código fonte usando um comando específico. O sistema pode ser instruído a abrir um editor arbitrário no arquivo e linha com base em seu banco de dados de origem. Ele vem com várias ferramentas gráficas e pode ser combinado com o SWI-Prolog Editor, PDT (plugin Eclipse para Prolog) ou GNU-Emacs.

- Compilador rápido:

  Mesmo aplicações muito grandes podem ser carregadas em segundos na maioria das máquinas.

- Flexibilidade:

  SWI-Prolog pode ser facilmente integrado com C. Ele também pode ser incorporado em programas externos. Predicados do sistema podem ser redefinidos localmente para fornecer compatibilidade com outros sistemas Prolog.

- Threads:

  Suporte robusto para vários threads pode melhorar o desempenho e é um fator chave para permitir a implantação de Prolog em aplicativos de servidor.

- Interfaces:

  SWI-Prolog vem com muitos pacotes de extensão que fornecem interfaces robustas para processos, criptografia, TCP / IP, TIPC, ODBC, SGML / XML / HTML, RDF, HTTP, gráficos e muito mais.

4.3 O Kit LEGO Mindstorms

O kit Mindstorms foi idealizado por Seymour Papert, um dos fundadores do MIT e autor da obra “Mindstorms: Children, Computers and Powerful Ideas”, onde demonstrou como computadores auxiliariam no desenvolvimento intelectual de crianças e jovens.

O LEGO Mindstorms é resultado de uma parceria entre o Media Lab do MIT e o LEGO Group, é a linha que une as famosas peças de montar da marca à tecnologia. Inicialmente com blocos de simples encaixe, o brinquedo evoluiu com o tempo passando pela linha Technic, cujos encaixes mais inteligentes permitiam movimentos de articulações, eixos e engrenagens, posteriormente sendo associados a motores, para finalmente permitir a participação de sensores e micro controladores. A Figura 4.1 mostra o kit na caixa.

Figura 4.1 Kit LEGO Mindstorms.

A LEGO lançou em 2006 os kits LEGO Mindstorms NXT. O kit conta agora com dois microprocessadores, um principal de 32 bits (AT91SAM7S256), com 48MHz e 256k de memória flash e 64k de memória RAM, e outro secundário de 8 bits (ATmega48), 4MHz e 4k de memória flash e 512b de RAM, um adaptador Bluetooth interno de 26MHz e uma entrada USB 1.1 (12Mbit/s), ambos permitindo conexão ao computador para transmissão de programas ou controle remoto do dispositivo.

4.3.1 Componentes do kit

Conforme é apresentado na Figura 4.2, o kit educacional conta com o brick (1), que tem entrada para três servo-motores e quatro sensores, os três servo-motores (6), um sensor

de distância por ultrassom(5), um sensor de luz (4), um sonoro (3) e dois de toque (2). A LEGO também disponibiliza em um kit complementar, mais peças para montar a estrutura do robô, além de sensores de movimentos como o Gyro e o Compass, ambos percebendo rotação em torno de um eixo, onde o primeiro percebe a velocidade de rotação e o segundo apenas a direção da rotação, e um sensor acelerômetro, que percebe aceleração nos três eixos ortogonais.

Figura 4.2 Componentes do Kit.

O sensor de som pode ser ajustado na escala decibel (dB) para detectar todos os sons com a mesma sensibilidade, em uma abrangência que inclui sons que extrapolam a sensibilidade do ouvido humano tanto para mais quanto para menos, e também na escala decibel alocada (dBA), este posicionando a largura de banda para padrões do ouvido humano.

As leituras são feitas em percentagens (%), sendo os menores níveis percentuais correspondentes a ambientes silenciosos. O sensor de toque é capaz de perceber quando é pressionado por algum objeto, e também quando é liberado. O sensor de luz possibilita ao robô perceber ambientes claros e escuros com uma escala de 0 a 100. Com a possibilidade de medir a reflexão luminosa de uma superfície, e como cada cor reflete a luz de uma forma diferente, o NXT torna-se capaz de distinguir cores, principalmente cores com tonalidades de preto e branco, onde a primeira indica ausência de luz (medidas baixas do sensor) e a segunda presença de luz (medidas altas).

O sensor ultra-sônico detecta a distância de obstáculos a até 255 centímetros com uma precisão de +/- 3 cm calculando o tempo de retorno de uma onda sonora refletida pelo objeto. Ele pode apresentar o resultado em centímetros ou em polegadas de acordo com a escolha do programador.

Os servo-motores podem funcionar como entradas e/ou saídas de dados. Eles funcionam como um motor de passos, sendo possível controlar o quanto o motor deve girar, e com qual velocidade ele deve fazê-lo em uma escala de 0 a 100. Cada motor possui também um sensor de rotações embutido, que permite a leitura dos movimentos do motor, podendo esta ser apresentada em graus, com precisão de +/- um grau e total para uma volta de 360 graus, ou em voltas completas, aplicada à medição de distâncias percorridas.

4.3.2 Estrutura utilizada

A estrutura adotada no projeto foi a Trike Base Figura 4.3 sugerida uma empresa que cria sensores extra para o NXT 2.0, a *HiTechnic*, pois é como um triciclo composto por duas rodas ligadas a servo-motores e uma roda em eixo livre. Essa arquitetura foi escolhida por ser de grande eficiência, pois com apenas dois servo-motores é possível controlar a direção e o sentido do deslocamento, obtendo-se os movimentos desejados a partir da utilização de um mínimo de estrutura. A essa estrutura básica foi acoplado o sensor de cor adequado para a proposta do projeto. O robô foi equipado com o sensor de cores adequado para a detecção de cores, cada cor representando uma informação proveniente do problema proposto, partindo da leitura deste sensor consulta a base de conhecimento e toma a decisão adequada seguindo os modelos da Lógica de Predicados.

Figura 4.3 Estrutura escolhida.

Devido à clara limitação do processador e da pequena memória do robô, foi necessário recorrer à ajuda de um computador para realizar o processamento das leituras do sensor, extraindo delas a informação correspondente ao problema e fazendo as consultas à base de conhecimento, que por sua vez retorna uma ação a ser realizada pelo robô em forma de um comando ou uma série de comandos.

4.4 Implementação

Nesta seção serão apresentados os detalhes da implementação de todo o programa de agente, tanto a definição da base de conhecimento, quanto da comunicação entre o computador e o robô LEGO.

Para este projeto foram necessárias algumas adaptações do problema original, devido a uma série de limitações do robô ou outras dificuldades técnicas. Uma delas é que, como não foi possível criar um objeto representando o Wumpus que deveria emitir um som ao ser atingido pela "flecha" e pela falta do sensor de som, a parte em que o agente dispara contra o Wumpus na tentativa de matá-lo foi desconsiderada. O agente nessa adaptação conhece de antemão o tamanho da caverna, pois ser fiel ao problema original neste aspecto causaria grandes desvios de locomoção do agente, locomoção que nem sempre tem precisão, sendo necessário interferir realizando ajustes manuais para corrigir sua posição, além disso, seria necessário adicionar mais dois sensores que aumentariam consideravelmente o peso do robô, aumentando consequentemente o consumo de bateria que tem pouca duração e se esgotaria muito mais rápido.

Para adequar o projeto à realidade, na modelagem do problema, a lista de percepções do agente foi reduzida para [nada, nada, nada], a primeira posição diz se o agente percebe fedor ou não, a segunda se ele percebe brisa e a terceira se percebe brilho. A quantidade de ações também foi reduzida, pois ele não pode disparar contra o Wumpus, e por uma questão de justiça o ouro não pode estar na mesma sala que o monstro ou um poço.

No problema original, cada vez que o agente se vira a noventa graus para uma direção, é atualizada a função de desempenho. Em nossa adaptação, por uma questão de simplicidade de modelagem e implementação, optou-se por não mapear estas ações na função de desempenho. Essa abordagem também é mais justa pois disponibiliza ao robô, que tem bateria com duração relativamente curta, mais ações sem penalidade na função de desempenho.

A função de desempenho está desmembrada pelo programa de agente, não existindo uma função específica para este fim.

Para separar melhor o que é um predicado daquilo que é regra a fim de facilitar a leitura do código, foram inseridos prefixos correspondentes a cada tipo de elemento. Os elementos que são predicados são precedidos da letra minúscula *p*, já as regras são precedidas da letra minúscula *r*. A parte restante dos nomes de símbolos seguiram a convenção conhecida como "*CamelCase*", onde cada palavra inicia-se com letra maiúscula. Por exemplo, no predicado *pAgenteEm*, o nome começa com o prefixo *p*, indicando que se trata de um predicado e não de uma regra, seguido da palavra *Agente* que por sua vez é seguida da palavra *Em*, ambas começando com letras maiúsculas. Com o propósito de organizar o código, a base de conhecimento foi dividida em arquivos diferentes, como se fossem módulos, cada um contendo apenas os predicados ou regras relacionados à parte da base de conhecimento a que ele se refere ou alguma regra geral necessária, como é demonstrado a seguir:

act.pl: este arquivo contém todos os predicados e regras relacionados as ações que o agente pode executar. Essas regras são apenas chamadas às regras que realmente implementam a ação, e foram necessárias para solucionar erros encontrados na base de conhecimento durante a execução do programa.

def.pl: arquivo que contém os predicados que definem o mundo como por exemplo o tempo.

imprime.pl: conjunto de regras auxiliares para imprimir na tela. Elas são muito úteis na hora de testar a base de conhecimento, pois com elas não é necessário que o robô esteja funcionando. Essas regras não serão detalhadas.

infer.pl: contém todos os predicados e regras necessário para que o agente realize a inferência a fim de descobrir onde estão os poços e o Wumpus.

init.pl: as regras contidas neste arquivo são responsáveis por inicializar corretamente a base de conhecimento do agente. Com o auxílio dos predicados *assert* e *retractall*, que são embutidos em Prolog, inconsistências são eliminadas desde o início da execução do programa de agente. Todas as sentenças são removidas e depois são reinseridas na base de conhecimento da maneira correta. Por exemplo, para evitar que o agente inicie a execução em outra sala que não a sala inicial, qualquer sentença sobre a localização do agente é

removida com *retractall*, e então a sentença que afirma que o agente está em [1,1] é inserida com *assert*.

percep.pl: este arquivo contém as regras que se encarregam da percepção do agente. Essas regras previnem que a percepção seja informada de forma equivocada ou não sejam informadas à base de conhecimento. Por exemplo, elas levam em consideração que o agente só pode perceber se estiver vivo, dessa forma não será informada nenhuma percepção quando ele não estiver.

regras.pl: conjunto de regras gerais, auxiliares e implementação das ações.

stratg.pl: contém as estratégias que o agente usa para decidir qual a ação deverá ser tomada com base nas percepções atuais e nas inferências que realizou.

test.pl: regras de simulação que auxiliam nos testes. Essas regras não serão abordadas.

utl.pl: regras utilitárias. Essas regras não estão diretamente relacionadas com o mundo de wumpus, porém são indispensáveis para o bom funcionamento da base de conhecimento. Por exemplo, a regra *exclude/3* é usada para excluir um elemento de uma lista de acordo com uma condição, dessa forma, é possível eliminar as salas seguras de uma lista de salas, caso seja necessário.

w.pl: arquivo principal responsável por carregar todos os outros arquivos.

4.4.1 Predicados de Definições

Os predicados de definições são todos dinâmicos através do predicado embutido "*dynamic*". Em Prolog, predicados dinâmicos são aqueles que podem mudar de maneira constante durante a execução. Nem todos os predicados deveriam ser dinâmicos, por exemplo, o predicado pBrisa/1, pois onde existe brisa continuará existindo até o final da execução, porém optou-se por tornar este e outros predicados em dinâmicos apenas por precaução.

pAdjacente/2: este predicado recebe uma sala e uma lista de salas como argumento. As salas são representadas por um par de coordenadas representado linha e coluna na forma [linha,coluna], por exemplo [1,1]. A lista de salas representam as salas que são vizinhas da sala que foi passada como argumento, exceto as salas diagonais.

pAgenteVivo/0: informa se o agente está vivo ou não. É utilizado para controlar as percepções, pois o agente não pode perceber se estiver morto, e também para controlar a execução do programa que se encerra se o agente morrer.

pDimensaoCaverna/1: este predicado determina o tamanho da caverna. É muito importante para definir a relação de adjacência entre as salas, pois sem ele não é possível definir os limites válidos para linha e coluna, nem se uma sala vizinha é diagonal ou não.

pTempo/1: determina e controla a passagem do tempo. O tempo é incrementado em um a cada ação do agente, sendo usado como referência a respeito de percepções e ações.

pAgenteEm/1: informa a localização do agente e é muito utilizado em várias regras, inclusive de inferência.

pLimiteTempo/1: tempo máximo de execução do teste, se for atingido o tempo máximo o teste é finalizado.

pSentidoAgente/1: indica para qual sentido o agente está voltado, norte, sul, leste ou oeste. É muito importante para a locomoção do agente.

pBrisa/1: este predicado é usada para informar que na sala *X*, o agente percebeu brisa, e portanto deve haver ao menos um poço por perto.

pFedor/1: este predicado é usada para informar que na sala *X*, o agente percebeu fedor, e portanto o Wumpus deve estar por perto.

pBrilho/1: usado para informar que o agente percebeu brilho na sala *X*, sendo assim encontrou o ouro.

pPossivelBuraco/1: este predicado serve para indicar quais salas possivelmente contêm um poço. Ele é imprescindível para inferir onde existem poços.

pPossivelWumpus/1: funciona da mesma forma que o predicado anterior, porém trata apenas das possíveis localizações do Wumpus.

pBuraco/1: quando o agente finalmente consegue inferir onde existe um poço ele usar este predicado para fazer tal afirmação. Por exemplo, ao afirmar (através de assert) *pBuraco(X)*, o agente está garantindo que ali existe um buraco.

pPontos/1: este predicado representa a pontuação acumulada do agente. Ela será incrementada ao encontrar ouro, ou decrementada ao morrer.

pSalaSegura/1: para tornar possível a tarefa de inferir onde existem poços ou o Wumpus é necessário saber, além de quais salas possivelmente são perigosas, quais são aquelas que certamente não oferecem risco. Para isso foi criado este predicado que afirma que determinada sala é segura.

pSalaVisitada/1: ajuda a tornar o agente mais explorador, evitando revisitar salas quando ainda existem salas que não foram visitadas. Também auxilia na inferência.

pAcao/2: predicado que é usado para informar a base de conhecimento qual ação foi realizada e quando foi realizada. Por exemplo, se a ação de "Mover" foi executada no tempo 3 (três), então será inserida esta informação na base de conhecimento, sob a forma "*pAcao('Mover', 3)*", para representar este fato.

pPercebe/3: controla qual percepção, onde e quando ela ocorreu. Por exemplo, o agente estava em [2,2], no tempo 4 (quatro) e percebeu brisa, então será informado este fato à base de conhecimento na forma "*pPercebe(brisa,[2,2],4)*".

pUltimaSala/1: informa qual foi a última sala a ser visitada, com exceção da sala atua, claro.

pWumpusEm/1: local onde o agente inferiu como sendo onde o Wumpus se encontra.

4.4.2 Regras de Ação

Regras responsáveis por informar a realização de ações à base de conhecimento, ou por chamar as regras que realmente executam esta tarefa. Foi necessário criar regras intermediárias como a regra *rExecAct*, para evitar erros que estavam ocorrendo durante a execução, pois surgiu a necessidade de ser capaz de fazer chamadas tanto à regras com argumentos, quanto regras sem argumentos.

rExecAct(Act,Arg): chama a ação contida em *Act*, passando os argumentos que estão em *Arg*.

rExecAct(Act): chama a ação contida em *Act*, que no caso não possui argumentos.

rActMove(X): executa a ação de mover-se, sempre para frente, até a sala *X*.

rActPegaOuro: executa a ação de "pegar" o ouro e atualiza a pontuação.

rActViraOeste: altera a orientação do agente para oeste.

rActViraLeste: altera a orientação do agente para leste.

rActViraNorte: altera a orientação do agente para norte.

rActViraSul: altera a orientação do agente para sul.

rActNada: dispensa explicações.

Todas as regras que mudam a orientação do agente geralmente antecedem a ação de mover-se até uma sala qualquer, e só executam se a direção do agente for diferente da sala de destino.

4.4.3 Regras de Inferência

As regras de inferências são a forma pela qual o agente é capaz de controlar a parte não observável do ambiente. São muito uteis para que ele consiga agir de forma racional atingindo seu objetivo de forma eficiente. É importante observar que uma regra por si só muitas vezes não é o suficiente. Para exemplificar, ambas as regras que tratam de descobrir onde existem poços ou o Wumpus não seriam capazes de cumprir com suas funções se não fosse o conhecimento de onde isso não é verdade, que é gerado por outras regras. Isso é uma consequência imediata da forma como a Lógica de Predicados utiliza os seus modelos a fim de determinar a verdade de cada sentença.

Os modelos onde salas seguras são ditas como perigosas são eliminados da base de conhecimento, através da inferência, pois agente não pode derivar conhecimento com premissas falsas. Restando então, apenas os modelos que são verdadeiros ou aqueles os quais não se pode afirmar nada.

rSalasSeguras: serve para inferir e informar à base de conhecimento quais salas são seguras, antes mesmo de visita-las. Quando o agente visita uma sala e não percebe brisa ou fedor, então todas as salas ao redor não podem conter poços ou Wumpus, sendo assim elas são seguras. Esta regra decorre da sentença lógica:

$$\forall x \neg Brisa(x) \wedge \neg Fedor(x) \Rightarrow \forall y\ Adjacente(x,y) \wedge Segura(y).$$

rPossivelWumpus: infere e informa à base de conhecimento em quais salas possivelmente o Wumpus está, sem a necessidade de visita-las. Quando existe a percepção de fedor em uma sala, isso significa que em uma das salas adjacentes está o Wumpus. A sentença lógica que originou está regra é:

$$\forall x\ Fedor(x) \Rightarrow \exists y\ Adjacente(x,y) \wedge PossivelWumpus(y).$$

rPossivelBuraco: infere e informa à base de conhecimento em quais salas possivelmente contêm um poço, sem a necessidade de visita-las. Quando existe a percepção de brisa em uma sala, é indício de que ao menos uma dessas salas contém um poço. Esta regra decorre da sentença lógica:

$$\forall x\ Brisa(x) \Rightarrow \exists y\ Adjacente(x,y) \wedge PossivelPoço(y).$$

rAchaBuraco: com esta regra o agente é capaz de inferir com exatidão se uma sala contem um poço. Primeiro é preciso utilizar a regra *rSalasSeguras* para descobrir e marcar quais salas são seguras, depois usando a regra *rPossivelBuraco*, o agente descobre quais são as salas passíveis de existir um poço. Por último, basta que o agente exclua as salas seguras do grupo de salas adjacentes, se sobrar apenas uma sala e esta for marcada como possível poço, então por inferência é lá que está o poço, caso ela não seja a única sala não é lógico afirmar nada. A seguir são apresentadas as sentenças lógicas que compões a regra:

$$\forall x\ Brisa(x) \Rightarrow \exists y\ possivelPoço(y) \wedge \neg \exists z\ Adjacente(x,z) \wedge PossivelPoço(z) \Rightarrow Poço(y).$$

rAchaWumpus: com esta regra o agente é capaz de inferir com exatidão onde o Wumpus está. Primeiro é preciso utilizar a regra *rSalasSeguras* para descobrir e marcar quais salas são seguras, depois usando a regra *rPossivelWumpus*, o agente descobre quais são as salas o Wumpus talvez esteja. Por último, basta que o agente exclua as salas seguras do grupo de salas adjacentes, se sobrar apenas uma sala e esta for marcada como possível esconderijo do Wumpus, então por inferência é lá que ele está, caso a sala não seja a única, então não é lógico afirmar nada. A seguir são apresentadas as sentenças lógicas que compõem a regra:

$$\forall x\ Fedor(x) \Rightarrow \exists y\ PossivelWumpus(y) \wedge$$

$$\neg \exists z\ Adjacente(x,z) \wedge PossivelWumpus(z) \Rightarrow Wumpus(y).$$

4.4.4 Regras e Inicialização

As regras de inicialização servem para garantir que a base de conhecimento será inicializada sem inconsistências, prevenindo futuros problemas durante a execução.

rInit: esta regra é responsável por garantir que a base de conhecimento seja inicializada de forma correta. Todos os predicados são retirados com o predicado embutido “retractall”. Logo após alguns predicados iniciais são inseridos como por exemplo o tamanho da caverna, o agente inicialmente em [1,1], etc.

rStart: chama rInit e uma regra para imprimir na tela.

4.4.5 Regras de Percepção

São responsáveis por construir e controlar as percepções do agente corretamente, evitando inconsistências.

rPercebe: é responsável por construir de forma correta a lista de percepções que e informa-la a base de conhecimento.

rFedor: auxilia a regra de percepção e informa a percepção de fedor em uma sala á base de conhecimento.

rBrisa: faz o mesmo que a regra anterior porém trata apenas da percepção de brisa.

rBrilho: atua da mesma forma que as duas anteriores, e ainda evita que a percepção de brilho seja informada novamente se o ouro já foi encontrado.

4.4.6 Regras Gerais

Muitas dessas são regras não estão envolvidas diretamente na resolução do problema, mas desempenham um papel auxiliar importantíssimo para que aquelas que se encarregam disso possam lograr êxito.

rFim: controla o fim da execução do programa. O programa será encerrado seja porque o agente morreu, porque o tempo limite foi atingido ou porque o ouro foi encontrado.

rSalaNaoVisitada: retorna a primeira sala não visitada na vizinhança, se ela existir.

rSalaSegura: retorna a primeira sala segura na vizinhança, se ela existir.

rSalaAleatoria: sorteia uma sala qualquer, exceto a última que foi visitada. É usada geralmente quando não há mais salas não visitadas e/ou seguras.

rSentidoAgente: determina com base no sentido atual, na sala atual e de destino, se o sentido do agente deve mudar para que ele possa mover-se até aonde pretende. Usa as coordenadas para concluir a necessidade da mudança. Por exemplo, o agente se encontra em [2,3], decidiu ir até [2,2] e está voltado para o leste, como [2,2] está mais a oeste, pois a coluna 2 é obviamente anterior a coluna 3, então a gente deve se voltar para oeste.

rAgenteStatus: retorna o status atual do agente, vivo ou morto.

rUpdatePontos: atualiza a pontuação acumulada com base no número passado como argumento. Se for negativo ocorrerá o decréscimo da pontuação, caso contrário acréscimo.

rUpdateAgenteStatus: responsável por verificar se o agente ainda está vivo, se morreu porque foi devorado ou caiu em um poço e informar à base de conhecimento.

rUpdateStatus: atualiza o conhecimento geral do agente, como o tempo, as informações de onde possivelmente há poços, etc., antes que ele tente tomar qualquer decisão. Isso evita que ele perca detalhes do ambiente e de suas mudanças, sendo mais preciso no processo de inferência.

rUpdateInfo e rInfo: utilizadas para atualizar e obter informações, para em seguida imprimir na tela.

4.4.7 Regras de Estratégia

As regras de estratégia determinam qual será o comportamento do agente face a percepção atual e o conhecimento construído até o momento. Visam imprimir um comportamento inteligente ao agente, evitando que este tome decisões que não decorram de um raciocínio derivado de sua base de conhecimento.

rEstrategia: esta regra é usada para definir qual ação deverá ser executada com base na lista de percepções recebida como argumento. Determina primeiramente, que o agente não pode decidir nada se estiver morto. Estabelece que o agente só deve visitar uma sala se ele não perceber nada que indique perigo, e que não faz muita diferença na ação a ser tomada quando ele percebe o perigo, pois o melhor é retornar de onde veio. O agente pode ainda, decidir não fazer nada.

4.4.8 Regras Auxiliares

São regras para processamento de listas, coisa que é muito utilizada em praticamente todo o código. Sem elas nada funcionaria corretamente, e muitas não são tão intuitivas de serem implementadas quanto parece.

filter: recebe um predicado, uma lista e uma variável, que também é uma lista, e filtra os elementos da lista que não unificam com o predicado, armazenando na variável apenas os que correspondem.

exclude: faz o contrário da regra anterior, exclui os elementos da lista que unificam com o predicado, armazenando na variável apenas os que não correspondem.

add: adiciona um elemento à cabeça de uma lista.

append: adiciona um elemento à calda de uma lista.

pertence: verifica se um determinado elemento pertence à lista

tamLista: usado para saber o tamanho de uma lista, ou seja, quantos elementos ela possui.

primeira: retorna a primeira posição da lista.

merge: une duas lista. Exemplo: *merge([a,b],[c,d],Lista)*, *Lista = [a,b,c,d]*.

4.5 Interface SWI-Prolog C++

Para controlar o agente e sincronizá-lo com o computador foi utilizado um pequeno programa escrito em C++. Essa linguagem de programação foi escolhida por dois motivos principais, o primeiro é que já existia uma interface definida do SWI-Prolog para ela, e segundo é que para utilizar outras linguagens como Java, seria necessário sobrescrever o firmware (sistema operacional) do robô, o que se feito incorretamente poderia danificá-lo permanentemente.

C++ fornece uma série de características que tornam possível definir uma interface natural e concisa para linguagens de tipagem dinâmica. Usando a conversão explícita de tipo (*casting*), tipos de dados nativos podem ser traduzidos automaticamente em tipos Prolog adequados, destrutores podem ser usados para lidar com a limpeza necessária e manipulação de exceção C ++ pode ser usada para mapear exceções Prolog e erros de conversão de interface para exceções C ++, que são automaticamente mapeados para exceções Prolog e o controle é devolvido para Prolog.

Variáveis Prolog são dinamicamente tipificadas e toda a informação é incorporada na classe *PlTerm*. Construtores e definições de operador fornecem operações e integração flexíveis com tipos C importantes (*char *, long, doubles*, etc).

4.5.1 Principais Classes

PlTerm: Esta classe é a base para a maioria das outras classes da interface, como *PlString*, *PlTerm* e *PlTermv* por exemplo. Com ela é possível criar termos Prolog que podem ser usados para consulta a partir de tipos de dados C. Desempenha um papel central na conversão e operação com dados do Prolog, pois ela fornece construtores e operadores para conversão de tipos C e verificação de tipos.

PlString: Subclasse de *PlTerm* com construtores de objetos *string* em Prolog. Um objeto dessa classe transforma *strings* C numa cadeia de caracteres que pode ser manipulada em Prolog, e vice-versa. Deve receber uma *string* C como parâmetro no seu construtor, mesmo que seja uma *string* vazia.

PlTermv: Esta classe é usada para criar vetores de termos Prolog. O operador "[ ]" está sobrecarregado para acessar elementos neste vetor, semelhante ao que acontece com vetores de inteiros em C. Os vetores são zero-indexados, ou seja, são indexados a partir de 0 (zero), dessa forma para acessar o primeiro elemento num vetor de termos Prolog basta fazer nomeVetor[0]. *PlTermv* também é usado para construir termos complexos e fornecer listas de argumentos para consultas Prolog.

PlTail: Subclasse de *PlTerm* para a construção e análise de listas Prolog. A classe *PlTail* é usada tanto para a análise quanto para construção de listas. É chamada *PlTail* porque a referência ao termo segue a "cauda" (Tail) da lista. Sua base é um vetor de termos Prolog, ou seja, um objeto da classe *PlTermv* que é passado como argumento para seu construtor.

PlCompound: Subclasse de *PlTerm* com construtores de termos compostos. Termos compostos são termos cujas partes são outros termos, assim o termo "*adjacente([1,1],[[1,2],[2,1]])*" é um termo composto. Para compor este termo é preciso utilizar tanto a classe *PlTerm* quanto a classe *PlTermv*. Primeiro declara-se dois objetos da classe *PlTerm* representado "adjacente" e "[1,2]" respectivamente, depois um *PlTermv* cria um vetor de termos para armazenar os elementos da lista de adjacência, este vetor será o núcleo de uma lista Prolog que pode ser criada com *PlTail*. Então compõe-se um o termo adjacente com *PlCompound*, passando o objeto que representa o termo "adjacente" e a lista Prolog como argumentos.

PlQuery: Encapsula as chamadas ao Prolog, representa a abertura e enumeração das soluções para uma consulta Prolog. O resultado da consulta utilizando esta classe é o conjunto de todas as soluções, tal qual acontece no interpretador Prolog, se houver mais soluções retorna *true*, caso contrário *false*. As unificações criadas pela consulta são persistentes e caso seja necessário desfazê-las para reutilizar as variáveis, deve-se utilizar a classe *PlFrame*.

PlCall: Cria uma *PlQuery* dos argumentos que recebe, a diferença é que desta vez ao invés de gerar o conjunto de todas as soluções, esta consulta gera apenas a primeira solução e a destrói. Retorna o resultado da solução ou uma exceção. Semelhante a *PlQuery*, *a*s

unificações criadas pela consulta são persistentes e para desfazê-las deve-se utilizar a classe *PlFrame*.

PlFrame: Esta classe utilitária pode ser usada para descartar referências não utilizadas, bem como fazer "retrocesso de dados". A classe *PlFrame* fornece uma interface para descartar referências a termo não utilizadas, bem como retroceder unificações. Esta classe é muito útil se o programa de principal é definido em C ++ e chama Prolog várias vezes. Criar argumentos para uma consulta requer referências a termo e utilizar *PlFrame* é a única maneira de recuperá-las. Um uso típico para *PlFrame* é a definição de funções C ++ que chamam Prolog repetidamente como laços *for* ou *while*.

PlEngine: Essa classe é usada em aplicações embarcadas, aplicações onde o controle principal é escrito em C ++. Ela fornece a criação e a destruição do ambiente Prolog. Na maioria das vezes, Prolog é a aplicação principal e C ++ é usado para adicionar funcionalidades ao Prolog, ou para acessar recursos externos ou por motivos de desempenho. Em algumas aplicações, é o contrário e deseja-se usar Prolog como um servidor de lógica. Por isso a classe *PlEngine* foi definida.

Apenas uma única instância desta classe pode existir em um processo. Quando usada em uma aplicação *multi-threading*, apenas um *thread* por vez pode ter uma consulta em execução. As aplicações devem garantir isso usando técnicas de "*lock*" adequadas.

4.6 Utilizando a Interface

A seguir será apresentada a forma como a interface SWI-Prolog C++ foi utilizada para tornar possível a comunicação bilateral entre o computador e o robô.

Em primeiro lugar é preciso iniciar o motor Prolog embutido na aplicação isso é feito como no trecho de código a seguir:

```
char* argv[] = {"swipl.dll", "-q -s" , "C:/w/w.pl" , NULL};
```

Sem isso o programa não seria capaz de utilizar o interpretador Prolog oferecido pelo SWI-Prolog.

As demais declarações e funções de chamadas ao Prolog estão listadas na sequência.

PlEngine e(3,argv): inicialização do interpretador Prolog com os parâmetros contidos em arg.

PlTerm Act, max: cria dois objetos da classe PlTerm cada um representando um termo Prolog.

PlTermv vetor(3): cria um vetor de três posições que armazenará as percepções do agente.

PlTail Lista(vetor[0]): usando o vetor criado anteriormente é criada a lista de percepções. A lista é preenchida quando uma *string* C++ é transformada em uma string C, depois é usada para construir uma lista Prolog utilizando a classe PlCompound.

PlString Executar: este objeto armazena a string Prolog que representa a regra que realiza as chamadas a regra *rExecutar* corretamente.

PlCall("rInit",0): chamada a regra de inicialização.

PlCall("rFim"): consulta a base de conhecimento a fim de saber se as condições que encerram o teste foram satisfeitas, caso positivo rFim retorna true, caso contrário false.

PlFrame fr: o objeto *fr* da classe *PlFrame* é responsável por permitir várias chamadas a regras e predicados Prolog repetitivamente. Sem ele, não seria possível desfazer uniões ou descartar referências a termos não utilizadas, o que impossibilitaria o programa de prosseguir.

string percep: armazena a lista de strings que representam as percepções do agente.

PlCall(Consulta,Teste): essa é a chamada ao Prolog que consulta a base de conhecimento perguntando qual ação deve ser tomada, dada a lista de percepções fornecida.

4.7 A Biblioteca NXT++

Para controlar o robô LEGO via Bluetooth, utilizando um programa escrito em C++, foram criadas diversas bibliotecas e interfaces que se propuseram a servir como um intermediário entre os dois. Algumas não foram selecionadas por exigirem a substituição do *firmware* do bloco de processamento do kit, outras por serem softwares corporativos e pagos.

A interface que se mostrou mais adequada ao projeto foi a NXT++, ela foi criada por Cory Walker e está disponível em https://github.com/corywalker/nxt-plus-plus na sua implementação original. Ela é livre e gratuita, o problema é que ela não contempla uma parte crucial do projeto, o sensor de cores. Então, foi necessário utilizar uma versão modificada da mesma interface, cujo responsável é Dr. Prof. Piotr Artiemjew, professor

assistente da Universidade de Warmia e Mazury em Olsztyn, Polônia, que pode ser baixada em http://wmii.uwm.edu.pl/~artem/nxtpp0-6-1.zip.

As maiorias das funções a seguir tratam da locomoção do agente, outras ajudam na coleta de dados para a construção correta da percepção que será enviada para a parte do programa de agente construída em Prolog. Existem ainda, funções responsáveis por criar e gerenciar a conexão Bluetooth.

Comm::NXTComm: declara um objeto da classe *Comm* que é responsável por criar, manter, operar e finalizar conexões Bluetooth.

NXT::OpenBT: estabelece uma conexão Bluetooth entre o robô e o computador.

NXT_Act: Essa função é a principal função C++ do projeto, pois ela é responsável por chamar a função responsável por executar a ação que retornou da consulta à base de conhecimento.

NXT_AcionaMotor: esta função aciona um dos motores com a potência especificada, fazendo-o girar o número de graus desejados. Se o número de graus for positivo a rotação será no sentido horário, caso contrário será no sentido anti-horário.

NXT_Frente: faz o robô ir para frente acionando os motores ao mesmo tempo no sentido horário, com a potência desejada e por um período de tempo definido durante a compilação.

NXT_MeiaVolta: acionado primeiro um motor no sentido anti-horário, depois o outro no sentido horário, faz o robô dar meia-volta. Logo após ajusta a posição ao chamar a função NXT_Frente.

NXT_MudaSentido: chama uma função auxiliar que verifica a necessidade de virar o robô antes de seguir para uma sala qualquer e, caso seja necessário, executa as ações de virar a esquerda ou direita, ou ainda meia-volta.

NXT_Stop: desliga ambos os motores.

NXT_GetColor: esta função é responsável por acionar o sensor de cor a fim de coletar informações que serão utilizadas por outra função para construir a percepção atual. Cada item da percepção é mapeado numa cor, sendo assim, ela verifica de qual cor se trata, pode ser azul representando uma brisa, amarelo que significa brilho, verde que representa o fedor do Wumpus ou vermelho que significa que o agente morreu. Após a verificação os

componentes da percepção são armazenados em strings que foram passadas como parâmetro.

NXT_GetPercep: com as *strings* fornecidas por NXT_GetColor, esta função monta a lista de percepções nos moldes de uma lista Prolog, ou seja, "[item 1, item 2, ..., item N]", porém como uma *string* C++ mesmo.

Cronometro: controla o tempo em que os motores ficam ligados.

NXT_GetSentido: verifica se o robô está numa direção diferente da sala para onde ele precisa se mover. Controla o sentido atual que está na base de conhecimento e o anterior que é o sentido físico do agente.

5 TESTES

Para proceder com os testes foram utilizados um notebook para executar o programa de agente e um adaptador Bluetooth para que fosse possível estabelecer uma conexão entre o robô e o notebook. O sistema operacional instalado no notebook é o Microsoft Windows 10, foram instalados o SWI-Prolog versão 6.5.2-136-g368476b, o Microsoft Visual Studio Ultimate 201 e o Phantom Driver em sua última versão que contém todos os drivers para o NXT 2.0.

O ambiente foi representado utilizando vários quadrados de papel cartolina de 30cm de lado aproximadamente, que representam as salas da caverna do mundo de Wumpus. Depois de alguns testes prévios, não foi encontrada uma forma de o agente ter múltiplas percepções, por exemplo, perceber brisa, fedor e brilho ao mesmo tempo, pois as leituras não eram precisas e condizentes com a realidade. O principal motivo é que o sensor de cor é capaz de perceber apenas preto, branco, azul, verde, vermelho e amarelo, dessa forma não é possível usa outra cor representando combinações de percepções, por exemplo, laranja igual a brisa e brilho. Por isso, cada quadrado possui apenas uma cor e o agente só é capaz de perceber uma situação de cada vez. A cor de cada quadrado deve ser capturada pelo sensor de cor do agente. Cada cor representa uma percepção diferente. Um quadrado azul significa uma sala onde o agente percebe a brisa que existe em salas adjacentes a um poço. Um quadrado verde representa uma sala com o fedor que o agente percebe nas salas vizinhas a sala onde está o Wumpus. Já o ouro é representado por um único quadrado amarelo, uma vez que o agente só percebe o brilho do ouro quando entra na sala onde existe ouro. Um quadrado vermelho foi usado para representar que o agente morreu, seja porque caiu em um poço ou porque entrou na sala do Wumpus. E por fim, um quadrado branco significa que o agente não percebe nada. Os quadrados podem ser arranjados de qualquer forma para caracterizar o mundo de Wumpus desde que respeite suas regras básicas como, por exemplo, a sala 1,1 não pode conter um poço nem o Wumpus, portanto o primeiro quadrado não pode ser vermelho.

Um obstáculo para proceder com os testes foi encontrar um ambiente com iluminação controlada, pois o sensor de cor pode sofrer interferência da luz ambiente e realizar leituras incorretas das cores. Também era necessário que este ambiente fosse espaçoso o suficiente para montar a “caverna” do mundo de Wumpus, já que a caverna ocupa uma área total de aproximadamente $(8 * 0{,}30\text{m})^2 = 5{,}76\text{m}^2$. A Figura 5.1 mostra

como fica o ambiente depois de montado.

Figura 5.1 Ambiente do mundo de Wumpus.

Várias configurações diferentes da caverna foram utilizadas nos testes, como mostra a Figura 5.2: um poço em 1,4 e outro em 4,4, o ouro em 2,2 e o Wumpus em 3,1. Depois de montar a "caverna", o agente é adicionado ao ambiente conforme as regras, ou seja, na sala 1,1 e voltado para o leste (direita). Diferente da notação clássica de coordenadas do mundo de Wumpus, a primeira coordenada é a linha e a segunda é a coluna, não o contrário. A Figura 5.2 mostra o agente físico na sala 1,1.

Figura 5.2 Posição inicial do agente.

Com o robô e o notebook ligados e a conexão Bluetooth estabelecida, o programa de agente deve ser executado para dar início aos testes. Primeiro o programa de agente

inicia a base de conhecimento, depois tenta estabelecer uma comunicação entre os dispositivos via Bluetooth, se bem sucedido ele faz a leitura da percepção atual e a primeira consulta à base de conhecimento, se a conexão falhar o programa é encerrado imediatamente. A Figura 5.3 mostra a tela quando a base de conhecimento é iniciada corretamente e a Figura 5.4 mostra a tela depois que a comunicação foi estabelecida e primeira leitura realizada.

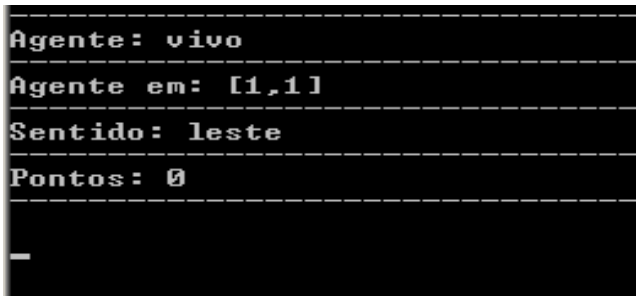

Figura 5.3 Tela inicial do programa de agente.

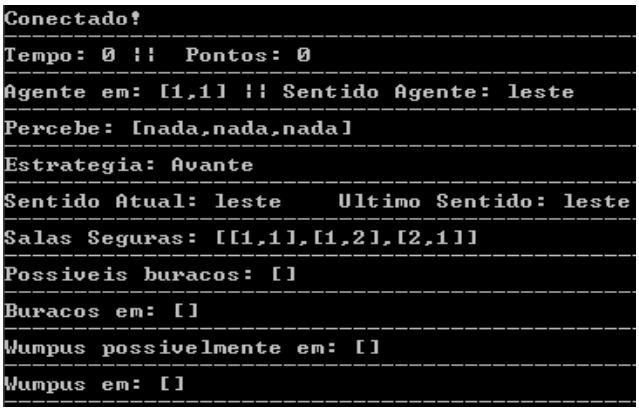

Figura 5.4 Tela do programa após a primeira leitura.

A partir da resposta oriunda da base de conhecimento, o agente executa uma ação, faz uma nova leitura do ambiente e informa a nova percepção à base de conhecimento através de uma nova consulta e assim por diante. O programa é encerrado quando o agente encontra o ouro, morre ou atinge o número máximo de ações executadas, que nos testes foi igual a 50.

Para exemplificar melhor; utilizando a configuração com um poço em 1,4, outro em 4,4, o ouro em 2,2 e o Wumpus em 3,1, a execução passo-a-passo de um teste é descrita em detalhes a seguir.

1. De acordo com as Figuras 5.2 e 5.4, o agente não percebe nada, pois se

encontra num quadrado branco. Tal percepção é informada à base de conhecimento por meio de uma consulta. A base de conhecimento usando as regras de inferência cria uma lista de salas que são seguras, no caso as salas vizinhas a 1,1. O agente usa suas regras de inferência para descobrir qual é a melhor ação a ser executada, dado o conhecimento do mundo até o momento e a percepção atual. Neste momento a ação selecionada foi avançar para a próxima sala não visitada e segura, no caso 1,2. A lista de salas seguras é atualizada pela base de conhecimento.

2. Após chegar a sala 1,2 como mostra a Figura 5.5, o agente executa a ação decorrente da inferência e faz uma nova leitura e consulta novamente a base de conhecimento. Como o agente continua não percebendo nada, a ação como era de se esperar é a mesma do passo anterior. Como mostra a Figura 5.6.

Figura 5.5 Agente em 1,2.

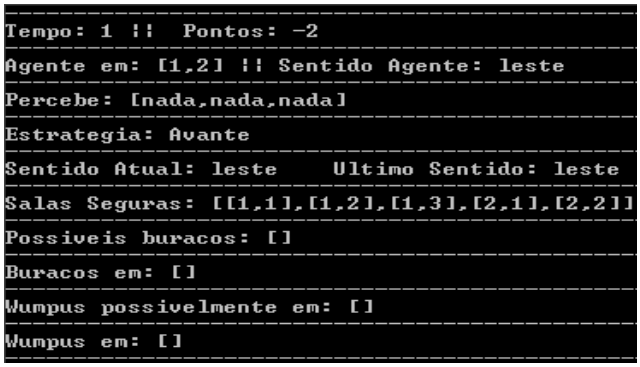

Figura 5.6 Tela do programa após a leitura em 1,2.

3. Agora na sala 1,3 como mostra a Figura 5.7, o agente percebe brisa, pois se encontra num quadrado azul, dessa forma a base de conhecimento utiliza as

regras de inferência para criar uma lista de quais salas são perigosas, ou seja, quais delas podem conter um poço. Conforme a Figura 5.8 as salas que possivelmente contem um poço são 1,4 e 2,3. Neste instante não é possível afirmar nada, a ação selecionada pela base de conhecimento foi se mover para uma sala segura.

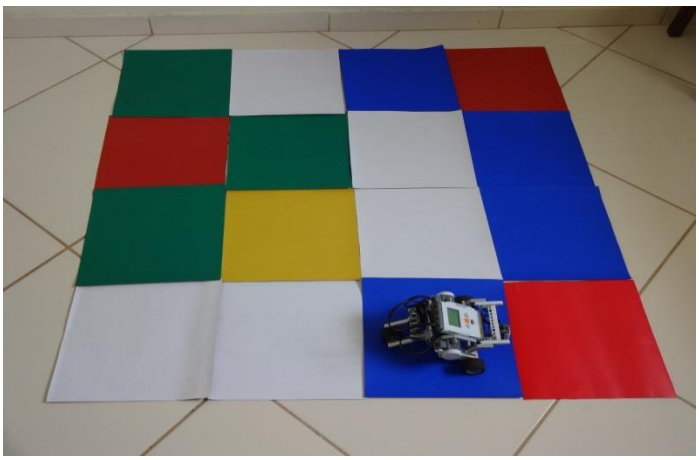

Figura 5.7 Agente em 1,3.

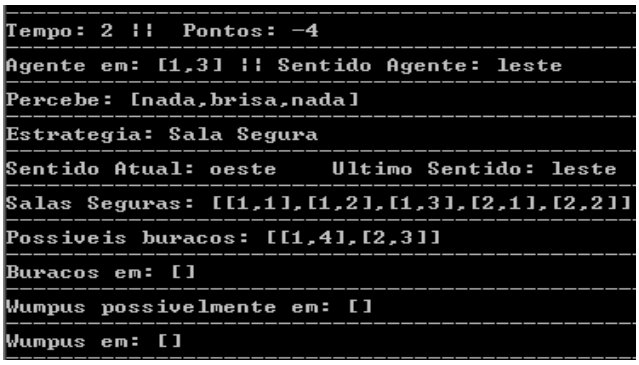

Figura 5.8 Tela do programa após leitura em 1.3

4. A única sala segura é 1,2, não há nenhuma razão especial para esta escolha, pois, se não há salas não visitadas ao redor, o agente vai se dirigir a uma sala segura aleatória. Após dar meia volta como mostra a Figura 5.9 o robô se dirige a 1,2. Como a sala 1,2 está representada por um quadro branco, novamente o agente não percebe nada. Portanto, ele deve se mover para uma sala não visitada, que no caso é 2,2.

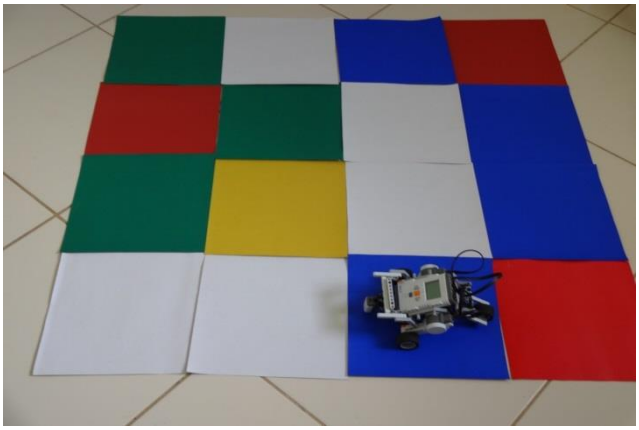

Figura 5.9 Agente após dar meia volta.

5. Agora o agente está em 2,2 e voltado para o norte. O quadrado representando a sala é amarelo o que indica que o ouro foi encontrado, então os pontos são atualizados adicionando 1000 e o teste é encerrado. A Figura 5.10 ilustra o este cenário, e a Figura 5.11 mostra a tela final do programa.

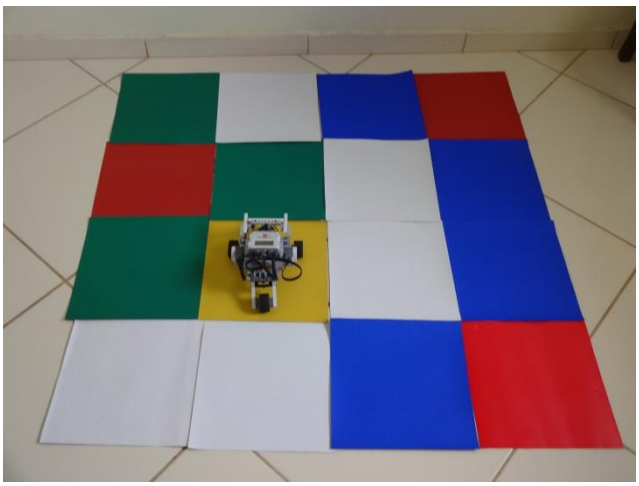

Figura 5.10 Agente encontra o "ouro" em 2,2.

```
Tempo: 4 ||  Pontos: 993
Agente em: [2,2] || Sentido Agente: oeste
Percebe: [nada,nada,brilho]
Estrategia: Pegar Ouro
Sentido Atual: oeste    Ultimo Sentido: norte
Salas Seguras: [[1,1],[1,2],[1,3],[2,1],[2,2],[3,1]]
Possiveis buracos: [[1,4],[2,3]]
Buracos em: []
Wumpus possivelmente em: [[2,3],[3,2]]
Wumpus em: []
```

Figura 5.11 Tela do programa após encontrar o "ouro"

Um teste mais complicado é realizado para verificar a capacidade de inferência do agente. Neste segundo teste o ambiente possui um poço em 1,4 e outro em 2,4, o ouro em 3,3 e o Wumpus em 3,1. A Figura 5.12 mostra a nova configuração do ambiente.

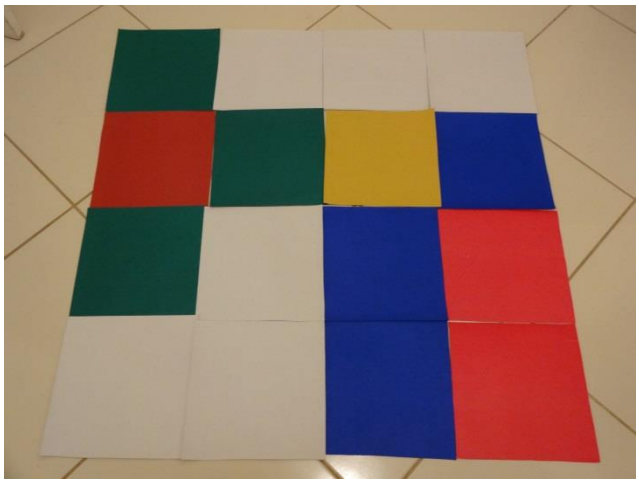

Figura 5.12 Configuração do segundo teste.

1. O agente está inicialmente na sala 1,1 e voltado para o leste (direita). A Figura 5.13 mostra o agente físico na sala 1,1. Como não há percepção que indique perigo, o agente se move para 1,2. As duas primeiras telas do programa são idênticas as do primeiro teste.

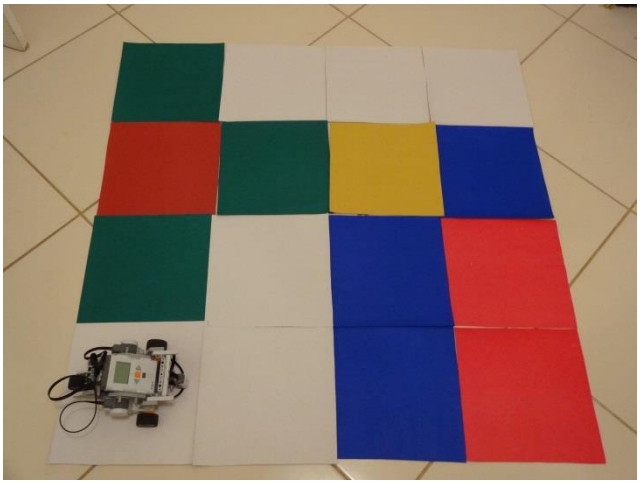

Figura 5.13 Posição inicial do agente (segundo teste).

2. Após chegar a sala 1,2 como mostra a Figura 5.14, o agente continua não percebendo nada, a ação é mover-se para 1,3.

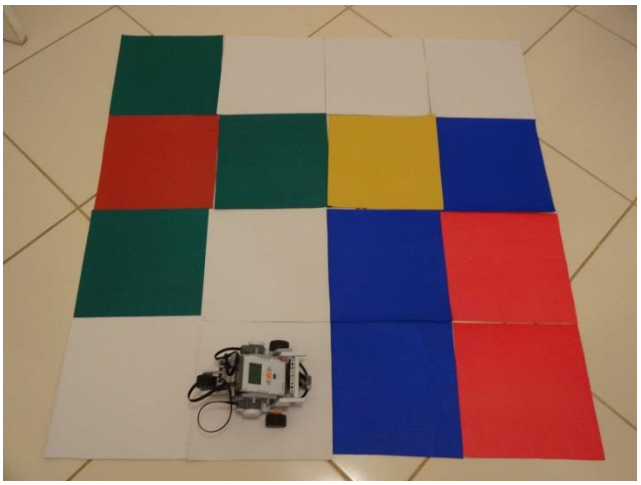

Figura 5.14 Agente em 1,2 (segundo teste).

3. Agora na sala 1,3 como mostra a Figura 5.15, o agente percebe brisa, pois se encontra num quadrado azul. Utilizando suas regras de inferência o agente cria uma lista de quais salas podem conter um poço. Neste instante não é possível afirmar nada, a ação selecionada pela base de conhecimento foi se mover para uma sala segura.

Figura 5.15 Agente em 1,3 (segundo teste).

4. De volta a é 1,2 que está representada por um quadro branco, novamente o agente não percebe nada. Portanto, ele deve se mover para uma sala não visitada, que no caso é 2,2. A Figura 5.16 mostra o cenário.

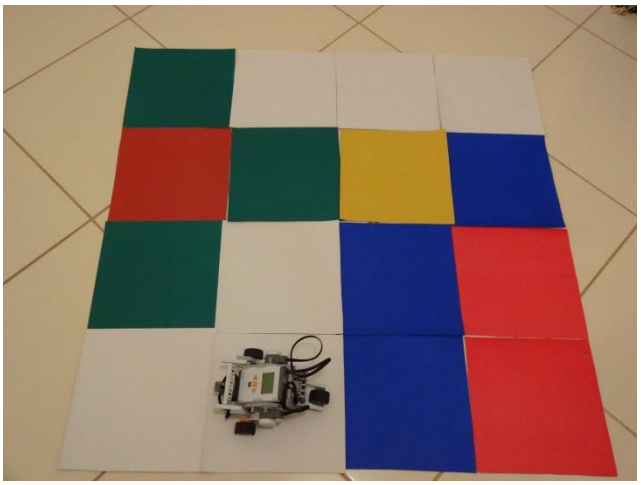

Figura 5.16 Retorno a 1,2 (segundo teste).

5. De acordo com a Figura 5.17, agora o agente está em 2,2 e voltado para o norte. O quadrado representando a sala é branco, o que indica que não há perigo na vizinhança. Assim o agente decide ir até 2,1.

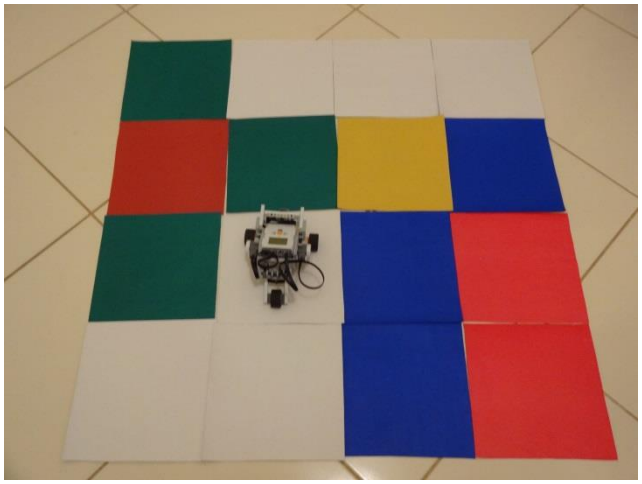

Figura 5.17 Agente em 2,2(segundo teste).

6. Neste momento o agente se encontra na sala 2,1, como ilustra a Figura 5.18, onde percebe o fedor do Wumpus, é um indicio de que o monstro está bem próximo. Quando o agente percebe o fedor, ele consulta uma lista de salas adjacentes, logo após são excluídas dessa lista as salas que reconhece como seguras, se restar apenas uma sala é lá que o monstro se encontra. Como as salas 1,1 e 2,2 são seguras, a única sala adjacente a 2,1 que resta é 3,1; logo o agente encontrou o Wumpus. O agente então decide voltar para 2,2

Figura 5.18 Agente em 2,1 (segundo teste).

7. Como 2,2 não indica perigo na vizinhança, o agente se move para 2,3 como mostra a Figura 2.3, onde percebe brisa. Neste ponto, o agente é capaz de saber que existe um poço em 1,4, pois 2,3 não é um poço. Além de adicionar 3,3 e 2,4 na lista de possíveis poços. Como mostra a Figura 5.19.

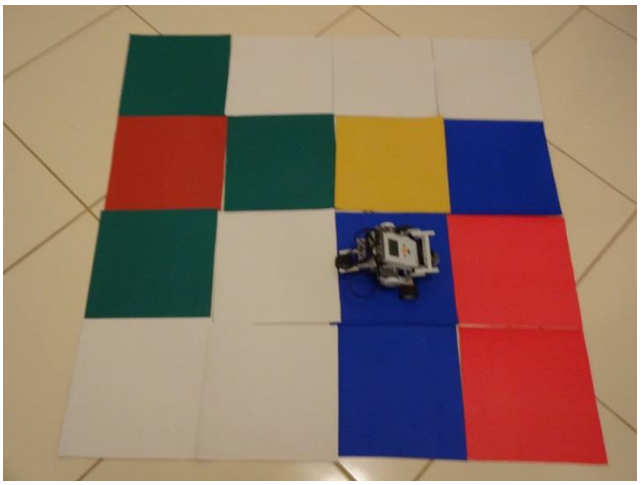

Figura 5.19 Agente em 2,3.

8. Novamente em 2,2, o agente resolve explorar uma linha acima se dirigindo para 3,2, como mostra a Figura 5.20. Nesta sala ele percebe o fedor do monstro, contudo não é preciso se preocupar com ele, pois o agente já inferiu que ele está em 3,1. Dessa maneira, o agente sabe que qualquer sala adjacente a 3,2 é segura, exceto 3,1, por isso decide ir até 3,3.

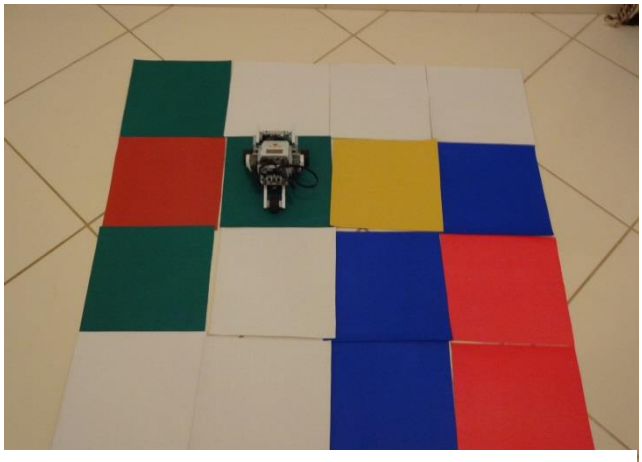

Figura 5.20 Agente em 3,2.

9. Em 3,3, como ilustra a Figura 5.21, o agente percebe o brilho do “ouro”, pois o quadrado é amarelo. Ele executa a ação de “pegar” o ouro, 1000 pontos são adicionados a sua pontuação atual e o teste é encerrado.

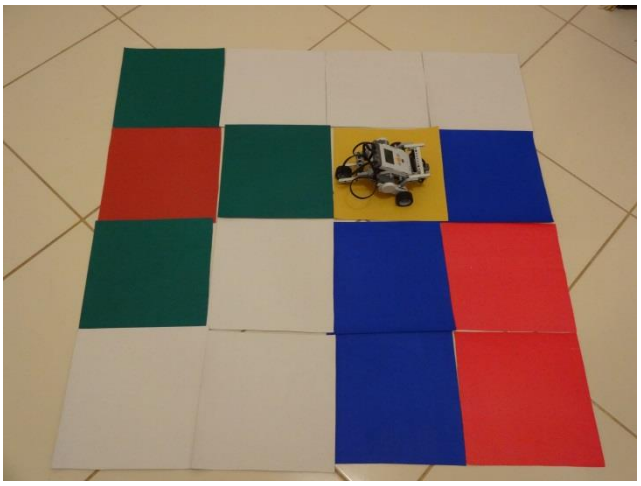

Figura 5.21 O agente encontra o "ouro" em 3,1.

Durante os testes foram identificados erros na contagem dos pontos, um problema no código fazia os pontos serem atualizados de forma incorreta às vezes. Outro problema encontrado é a imprecisão no movimento do robô, as funções fornecidas pela interface NXT++ não têm a precisão para manter o agente alinhado. A leve variação entre as medidas dos quadrados e a variação na potência total das baterias também interferiram um pouco na precisão do movimento. Por conta dessas imprecisões, foram necessárias várias intervenções para corrigir a posição do agente, evitando leituras incompatíveis com a realidade e que o agente se “perdesse” no ambiente.

A maioria dos testes terminou com o agente encontrando o ouro, outros encerraram ao atingir o número máximo de ações. Em certos testes, o agente não foi capaz de tomar nenhuma decisão, por exemplo, se ao iniciar o teste logo na primeira sala (1,1), o agente perceber brisa ou fedor, ele é incapaz de decidir sobre o que fazer, uma vez que não consegue inferir onde pode estar o perigo, pois não consegue coletar informações já que não pode se arriscar avançando para 1,2 ou 2,1. Muito poucos testes terminaram por falta de bateria. A média da pontuação foi de 930 pontos, essa pontuação muito próxima da recompensa por achar o ouro, acontece porque os testes são completamente independentes, em outras palavras, a execução e a pontuação do teste anterior não interferem na execução do teste atual.

6 CONCLUSÃO

Embora os diferentes tipos de agente possuam características em comum, é preciso um cuidado muito grande ao escolher qual tipo de agente deve ser utilizado no projeto. Um agente reativo simples não seria capaz de eliminar da base de conhecimento as configurações do mundo que não conferem com a realidade, pois ele não possui um mecanismo que o permite tomar tal decisão.

O agente baseado em modelos se mostrou adequado para o problema proposto no Mundo de Wumpus, pois ele possui o mecanismo de inferência que o permite explorar um ambiente de tarefas parcialmente observável. Em decorrência disso, durante os testes, ele foi capaz de, na maioria das vezes, descobrir onde estavam o ouro e/ou o perigo, seja um poço ou o Wumpus.

Utilizar um mecanismo para lidar com a incerteza poderia tornar o agente ainda mais preciso e menos limitado, considerando que em alguns momentos ele não é capaz de reunir informações suficientes para inferir nada sobre o ambiente, uma vez que sua medida de desempenho o impede de arriscar-se e, caso logre êxito, explorar uma parte que não era observável, a despeito do risco de morrer.

O Mundo de Wumpus é um problema da área de Inteligência Artificial que apesar de simples é muito didático, principalmente em relação ao aprendizado de projeto de agentes inteligentes, Lógica de Predicados.

A Lógica de Predicados encontra no Mundo de Wumpus um ótimo ambiente de aplicação e aprendizado, pois as regras de inferência contidas na função agente exigem a utilização de vários conceitos da Lógica de Predicados como quantificadores existenciais e universais, variáveis, símbolos, termos, predicados e regras. São os mecanismos pertencentes à Lógica de Predicados que permitem ao agente fazer inferências eliminando modelos falsos da base de conhecimento.

Compreender e utilizar o Paradigma de Programação Lógica é um grande desafio para aqueles acostumados ao paradigma imperativo. O programador é obrigado a mudar completamente sua forma de pensar, ao invés de pensar em termos de como resolver o problema, ele é desafiado a pensar em termos de como é o problema. O que antes era resolvido com algumas variáveis, laços de repetição e controle de fluxo, agora deve ser totalmente declarativo e fazer uso de recursão constantemente.

Unir C++ e Prolog no mesmo programa nem sempre é uma tarefa fácil, ainda que exista uma interface consolidada entre as duas como a que é fornecida pelo SWI-Prolog, muitas vezes a documentação é escassa, pouco didática e abrangente. Tipos de dados C++ podem ser traduzidos automaticamente em tipos Prolog, e o tratamento de exceções C++ é mapeado diretamente para Prolog. Mesmo assim, algumas exceções são de difícil compreensão e, como foi dito a documentação não ajuda muito. Apesar de tudo, a compatibilidade entre as duas linguagens é grande, o que permite contornar muitos obstáculos durante a implementação.

O kit LEGO Mindstorms NXT 2.0 é excelente para o aprendizado em inteligência artificial. Os vários modelos de arquitetura disponíveis facilitam sua aplicação numa gama de problemas que podem ser simulados num ambiente real. Embora possua poder de processamento e memória limitados, com a ajuda da biblioteca NXT++ é possível delegar essas tarefas a um computador via conexão Bluetooth, além de controlar os motores e sensores do robô.

A aplicação das várias ferramentas conceituais de software e hardware utilizadas neste projeto, a saber: o paradigma de programação lógico, Prolog, C++, o kit LEGO Mindstorms, a biblioteca NXT++, o Visual Studio Professional Ultimate e o SWI-Prolog, se mostram muito úteis para o aprendizado e aplicação prática nas várias áreas que elas se inserem, como programação, inteligência artificial e robótica.

6.1 Trabalhos Futuros

A seguir são apresentados alguns trabalhos futuros a serem realizados.

- **Reorganização do código Prolog:** a base de conhecimento foi bem estruturada e definida, contudo é preciso fazer algumas mudanças a fim de refinar o código. Por exemplo, a medida de desempenho encontra-se desmembrada em vários trechos de código, é desejável que ela se torne uma regra como um bloco de código de fato. Algumas regras são bem parecidas diferindo apenas pela aridade, ou seja, o número de argumentos que recebe, sendo assim, é preciso encontrar uma implementação mais clara de tais regras.
- **Estender o mecanismo de inferência:** é possível expandir a capacidade do mecanismo de inferência do agente para lidar com a incerteza. Atualmente o agente só é capaz de inferir se possui informações o suficiente, obtidas através da exploração do ambiente. Caso ele não consiga explorar certa parte do ambiente, não será capaz de inferir nada a respeito dela.
- **Aprimoramento da locomoção do agente:** a locomoção do agente às vezes é muito imprecisa, um problema que tem origem no próprio sistema do kit LEGO Mindstorms segundo Cory Walker. Por isso, é necessário ajustar sua posição no ambiente para evitar que ele se perca ou que faça uma leitura incorreta. Encontrar uma forma de eliminar este imbróglio tornaria o projeto mais elegante, pois o agente seria completamente autônomo sempre.

7 REFERÊNCIAS BIBLIOGRÁFICAS

L. Sterling and E. Shapiro. The Art of Prolog, MIT Press, 1994.

ECKEL, B. Thinking in C++, 2a ed. Prentice Hall, 1999.

WIELEMAKER, J. SWI-Prolog 5.6 Reference Manual, Disponível em: <http://www.swi-prolog.org/download/stable/doc/SWI-Prolog-5.6.59.pdf>. Acesso em: 05 dez 2015.

WIELEMAKER, J. A C++ interface to SWI-Prolog, Disponível em: <http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/pl2cpp.html%27)>. Acesso em: 05 dez 2015.

Russel, Stuart e Norvig, Peter. Inteligência Artificial, Elsevier, 2004.

Biblioteca NXT++, Disponível em <https://github.com/corywalker/nxt-plus-plus>. Acesso em: 05 de mar 2016.

Biblioteca NXT++ modificada, Disponível em <http://wmii.uwm.edu.pl/~artem/nxtpp0-6-1.zip>. Acesso em: 05 de mar 2016.

ANEXO A – INTERFACE SWI-PROLOG E C++ NA PRÁTICA

INTRODUÇÃO

C++ fornece uma série de características que tornam possível definir uma interface natural e concisa para linguagens de tipagem dinâmica. Usando a conversão de tipo (casting), tipos de dados nativos podem ser traduzidos automaticamente em tipos Prolog adequados, destrutores podem ser usados para lidar com a limpeza necessária e manipulação de exceção C ++ pode ser usada para mapear exceções Prolog e erros de conversão de interface para exceções C ++, que são automaticamente mapeados para exceções Prolog e o controle é devolvido para Prolog.

O objetivo é abordar a interface entre SWI-Prolog e C++, suas principais classes e seus operadores, conversões de tipos de dados, consultas a predicados e regras definidos em arquivos Prolog, bem como o uso intuitivo e natural de todos estes recursos através de exemplos simples previamente testados.

Todo este esforço se deve ao fato de que não há documentação sobre o assunto quando se trata de C++ como o programa principal e Prolog embarcado.

CONVENÇÕES

Antes de continuar é preciso esclarecer algumas convenções que serão utilizadas. Segue abaixo a lista de convenções:

- Assume-se que o leitor possui conhecimento de programação, utilizando tanto C++ quanto Prolog.
- Nem todo o conteúdo sobre interface será abordado, somente o que é mais didático para qualquer interessado que deseje aprendê-la.
- O controle principal da aplicação é definido em C++, que fará chamadas ao Prolog.
- Considere que todas as declarações como inclusão de bibliotecas, diretivas de processador e declarações de funções *main* já foram feitas.
- A intenção dos exemplos é mostrar o uso mais intuitivo, e fácil possível sobre cada ponto.
- Nas definições de métodos, funções, operadores e etc, o retorno e os argumentos estão em *itálico*, já os seus nomes estão em **negrito**, tipos de dados também estão em *itálico*.

PRINCIPAIS CLASSES

A área mais útil para a exploração de recursos C ++ é a conversão de tipo, variáveis Prolog são dinamicamente tipificadas e toda a informação é incorporada na classe PlTerm. Construtores e definições de operador fornecem operações e integração flexíveis com tipos C importantes (*char* *, *long*, *doubles*, etc).

PlTerm

A classe PlTerm desempenha um papel central na conversão e operação com dados do Prolog. Esta seção fornece documentação dos principais pontos dessa classe.

Construtores

PlTerm::PlTerm()

Cria um novo termo inicializado(contendo uma variável Prolog).

Exemplo:

```
PlTerm TermoGenerico;
```

PlTerm::PlTerm (*const char* *)

Cria uma referência a termo que contem um átomo Prolog representando texto.

Exemplo:

```
PlTerm Termo("aviao");
```

PlTerm::PlTerm (*const PlAtom &*)

Cria uma referência a termo que contem um átomo Prolog.

Exemplo:

```
PlAtom atomo("frutas");
PlTerm Termo(atomo);
```

PlTerm::PlTerm (*long n*)

Cria uma referência a termo que contem um número inteiro Prolog representando *n*.

Exemplo:

```
long int numero=200;
PlTerm Longo(numero);
```

PlTerm::PlTerm(*double f*)

Cria uma referência para termo contendo um *float* Prolog representando *f*.

Exemplo:

```
double tDouble= 375555;
PlTerm TermDouble(tDouble);
```

Operadores

Operadores de Conversão.

Um objeto PlTerm pode ser convertido para tipos C++ e vice-versa através dos seguintes operadores:

PlTerm::operator long (*void*)

Produz um inteiro longo se o PlTerm é um inteiro Prolog ou flutuante que pode ser convertido sem perda para um longo.

Exemplo:

```
long numero=(long)Termo;
```

PlTerm::operator int(*void*)

O mesmo que o anterior, mas pode representar menos bits.

Exemplo:

```
int numero=(int)Termo;
```

PlTerm::operator double (*void*)

Produz um *double* C se PlTerm representa um número Prolog inteiro ou *float*.

Exemplo:

```
double numero=(double)Termo;
```

PlTerm::operator char * (*void*)

Converte átomos e strings Prolog para o texto representado.

Exemplo:

```
char *texto=(char *)Termo;
```

Operador de Unificação

Este operador está definido para os tipos PlTerm, *long*, *double*, *char* *, PlAtom, entre outros. Ele executa uma unificação e retorna 1 (*true*) se bem-sucedido e 0 (*false*) caso contrário.

int **PlTerm::operator**=(*tipo*)

Exemplo:

```
PlTerm texto;//texto contem uma variável e por isso
texto="atribuir";//unifica com qualquer valor de tipo válido
if (texto = "unificar")//verificando se os termos se unificam
        cout<<"\nOs termos unificao.\n";
else
        cout<<"\nOs termos nao unificao.\n";
```

Ao contrário do que se pode pensar, na terceira linha de código, "*texto*" não receberá a string "*unificar*", Prolog apenas verifica se os dois termos unificam.

Operadores de Comparação

A comparação realizada por estes operadores é um pouco diferente, além de comparar o valor contido no objeto, é levado em conta, em primeiro lugar inclusive, a natureza interna ao Prolog dos termos que estão sendo comparados. Por exemplo, mesmo que um objeto PlString e um PlTerm contenham o mesmo texto, se forem comparados pelo operador "==" o resultado será que eles são diferentes entre si. De resto, o funcionamento dos operadores de comparação seguem a mesma lógica dos operadores C++ padrão correspondentes.

int **PlTerm::operator** ==(*const PlTerm &*)

Exemplo:

```
PlTerm Texto1, Texto2;
Texto1="Teste";
Texto2="operador";
//testando a igualdade pelo retorno do operador ==
if(Texto1==Texto2;)
        cout<<"\nOs termos são iguais\n";
else
        cout<<"\nOs termos são diferentes\n";
```

int **PlTerm::operator** != (*const PlTerm &*)

Exemplo:

```
PlTerm Texto1, Texto2;
Texto1="Teste";
Texto2="operador";
//testando a diferença pelo retorno do operador !=
if(Texto1!=Texto2;)
        cout<<"\nOs termos são diferentes\n";
else
```

```
cout<<“\nOs termos são iguais\n”;
```

int **PlTerm::operator** <(*const PlTerm &*)

Exemplo:

```
PlTerm Numero1, Numero2;
Numero1= 2;
Numero2= 1;
//testando a desigualdade pelo retorno do operador <
if(Numero2<Numero1)
        cout<<"\nNumero2 e menor que Numero1\n";
else
        cout<<"\nNumero2 nao e menor que Numero1\n";
```

int **PlTerm::operator** >(*const PlTerm &*)

Exemplo:

```
PlTerm Numero1, Numero2;
Numero1=2;
Numero2=1;
//testando a desigualdade pelo retorno do operador >
if(Numero1>Numero2)
        cout<<"\nNumero1 e maior que Numero2\n";
else
        cout<<"\nNumero1 nao e maior que Numero2\n";
```

int **PlTerm::operator** <=(*const PlTerm &*)

Exemplo:

```
PlTerm Numero1, Numero2;
Numero1=20;
Numero2=20;
if (Numero1<=Numero2)//comparando pelo retorno do operador <=
        cout<<"\nNumero1 e menor ou igual a Numero2\n";
else
        cout<<"\nNumero1 e maior que Numero2\n";
```

int **PlTerm::operator** >=(*const PlTerm &*)

Exemplo:

```
PlTerm Numero1, Numero2;
Numero1=25;
Numero2=20;
if (Numero1 >= Numero2)//comparando pelo retorno do operador >=
        cout<<"\nNumero1 e maior ou igual a Numero2\n";
else
        cout<<"\nNumero1 e menor que Numero2\n";
```

Os operadores citados acima estão sobrecarregados, sendo definidos também para argumentos do tipo long int. *Quando um* long int *é passado como argumento, PlTerm é convertido para este tipo e uma comparação C padrão entre dois inteiros longos é realizada, retornando 1 (*true*) se bem-sucedido ou 0 (*false*) caso contrário.*

int ***PlTerm::operator*** *==(*const char *texto*)*

*Retorna 1(*true*) se PlTerm é um átomo ou uma string representando o mesmo texto como argumento e 0 (*false*) caso contrário.*

Exemplo:

```
PlTerm Texto;
Texto="texto";
if(Texto=="diferente")
        cout<<"\nO termo e igual a constante.\n";
else
        cout<<"\nO termo e diferente da constante.\n";
```

Operador name

const char * PlTerm::name*()*

O operador name *retorna o nome do termo como um* char *, e está definido para todas as classes exceto para PlTermv, PlString e PlAtom.*

Exemplo:

```
PlTerm Aluno, Professor;
Aluno="joao";
Professor="maria";
PlTermv vetor(2);
// preenchendo o vetor
vetor[0]= Aluno;
vetor[1]= Professor;
//imprimindo o nome dos termos
cout<<'\n'<<(char *)vetor[0].name()<<'\n';
cout<<'\n'<<(char *)vetor[1].name()<<'\n';
```

Operador arity

int PlTerm::arity*( )*

Retorna a aridade de um termo, ou seja, o número de argumentos que ele recebe. Está definido para todas as classes exceto para PlTermv e PlAtom.

Exemplo:

```
PlTerm Aluno;
Aluno="joao";
cout<<'\n'<<Aluno.arity()<<'\n';
```

Operador type

int PlTerm::type*( )*

Fornece o tipo real do termo, retornando um inteiro para cada tipo começando de 1, os tipos são PL_VARIABLE, PL_FLOAT, PL_INTEGER, PL_ATOM, PL_STRING ou PL_TERM. Não está definido para PlAtom.

Exemplo:

```
PlTerm Termo;
PlAtom atomo= 2.55545455466;//PlAtom será abordada na próxima seção
Termo= atomo;
cout<<"Tipo: "<< Termo.type()<<'\n';
```

PlAtom

Permite a manipulação de átomos Prolog em sua representação interna para comparação rápida.

Construtor

PlAtom::PlAtom(const char *texto)

Cria um átomo a partir da string. Internamente ao Prolog, a representação do átomo pode não ser um texto e sim um número inteiro por exemplo.

Exemplo:

```
PlAtom atomo="ancestral";
cout<<'\n'<<(char *)atomo;
//na linha a seguir o inteiro 200 será convertido para texto, apenas para
//criar o átomo, mas internamente ao Prolog ele será um inteiro.
PlAtom OutroAtomo= 200;
cout<<'\n'<<(char *)OutroAtomo;
```

PlString

Subclasse de PlTerm com construtores para a construção de objetos string em Prolog. Uma string no SWI-Prolog representa uma *byte-string* na pilha global. Seu tempo de vida é o mesmo que para termos compostos e outros dados que estão na pilha global. Strings não são apenas uma representação composta de texto que é coletado como lixo,

mas como elas podem conter *0-bytes*, podem ser usadas para conter estruturas de dados C arbitrárias.

Construtores

PlString::PlString(const char * texto)

Cria um objeto string SWI-Prolog a partir de uma string C. O texto é copiado.

Exemplo:

```
PlString Nome("Joao");
cout<<"\nNome: "<<(char *)Nome<<'\n';
```

PlString::PlString*(*const char *texto, size_t comprimento*)*

*Cria um objeto string SWI-Prolog a partir de uma string C com o comprimento especificado. O texto pode conter carácter "\0" (*null*) e é copiado.*

Exemplo:

```
char* nome;
nome="Automovel";
PlString String(nome, 4);
cout<<"\nNome: "<<(char *)String<<'\n';
```

PlTermv

Vector de termos Prolog. O operador [ ] está sobrecarregado para acessar elementos neste vetor. PlTermv é usado para construir termos complexos e fornecer listas de argumentos para consultas Prolog.

Operador [ ]

PlTerm PlTerm::operator**[ ]***(*int indice*)*

Exemplo:

```
PlTerm Aluno, Professor;
Aluno="paulo";
Professor="maria";
PlTermv vetor(2);
```

```
vetor[0]= Aluno; //preenchendo o vetor
vetor[1]= Professor;
cout<<'\n'<<(char *)vetor[0]<<'\t'<<(char *)vetor[1]<<'\n';
```

PlTail

Subclasse de PlTerm para a construção e análise de listas de Prolog. *A classe PlTail é usada tanto para a análise quanto para construção de listas. É chamada PlTail porque a referência a termo segue a "cauda" (Tail) da lista.*

Operadores

int PlTail::append*(*const PlTerm &elemento*)*

Acrescenta um elemento à lista e faz a referência a PlTail apontar para a nova variável de cauda. Se A é uma variável dentro de uma lista, e essa função é chamada com o argumento "carro", uma lista na forma [carro | B] é criada e objeto PlTail agora aponta para a nova variável B.

Exemplo:

```
PlTermv vetor(1);
PlTail Lista(vetor[0]);
string entrada;
for(int i =0; i < 5; i++){
	cout<<"\nDigite uma palavra: ";
	cin>>entrada;
	Lista.append(entrada.c_str());
}
Lista.close(); //a função a seguir será explanada mais adiante,
PlCall("writeln",vetor[0]); //chamada ao predicado writeln de Prolog
```

int PlTail::next(PlTerm &elemento)

Liga "elemento" ao próximo elemento da lista e avança. Retorna 1 (*true*) em caso de sucesso e 0 (*false*) se PlTail representa a lista vazia. Se PlTail não é nem uma lista nem uma lista vazia, um erro de tipo é lançado.

Para o exemplo a seguir primeiro é preciso construir uma lista, o trecho de código abaixo faz isso ao construir uma string, usando concatenação, na forma "[*elemento1, elemento2, ..., elementoN*]" que é a forma padrão de listas em Prolog. Vale ressaltar que se o padrão Prolog de listas não for seguido, a lista não será válida. Exemplo de lista valida: [*a, b, c*], exemplo de lista inválida: [*a, b, c,*] ou *a, b, c*.

Exemplo:

```
PlTerm elemento;
PlTermv vetor(5); //tamanho pré-definido de lista
PlTail Lista(vetor[0]);
string entrada, componentes="["; //iniciando a lista com "["
for(int i =0; i < 5; i++){
	cout<<"\nDigite uma palavra: ";
	cin>>entrada;
	componentes+=entrada;//concatenando a string com a entrada
	if(i!=4)//evitando quebrar o padrão ex: [a,b,c,] resulta em erro
		componentes+=", ";
	cin.clear();
	entrada.clear();
}
componentes+="]";
Lista=(PlCompound)componentes.c_str();
while(Lista.next(elemento))
	cout<<(char *)elemento<<' ';
```

int PlTail::close( )

Unifica a representação interna da lista com [d] e retorna o resultado da unificação. Essa função foi usada quando uma lista foi construída utilizando-se o operador *append*, ela "fechou" a lista, caso contrário a calda da lista continuaria contendo uma variável. Para visualizar seus efeitos comente a linha de código referente a esta função no exemplo do operador *append*.

PlCompound

Subclasse de PlTerm com construtores para a construção de termos compostos.

Construtores

PlCompound::PlCompound(const char *texto*)*

Cria um termo através da análise do texto. Se o texto não está na sintaxe Prolog válida, uma exceção é lançada. Caso contrário uma nova referência a termo contendo o texto analisado é criada.

Exemplo:

```
string imprime="[banana, abacaxi, morango]";
//construindo a estrutura da lista de argumentos
PlTermv vetor(3);
PlTail Lista(vetor[0]);
//criando uma lista de argumentos com o termo composto
```

```
//c_str converte uma string em char*
Lista= (PlCompound)imprime.c_str();
PlCall("writeln",Lista);
```

PlCompound::PlCompound(const char *nome_termo, PlTermv argumentos)

Cria um termo composto com o nome dado e os argumentos contidos no vetor.

Exemplo:

```
PlTermv argumentos(2), variavel(1);
PlTail lista(argumentos[0]);
argumentos[1]="world"; //este será o argumento do termo composto
//Criando o termo composto hello(world)
PlCompound Composto("hello",argumentos[1]); //"hello" será o nome do temo
//Fazendo chamadas ao Prolog,
//usa-se nesse caso "assert" pelo mesmo motivo que se usaria em Prolog,

//ou seja, é preciso afirmar que o termo existe.
PlCall("assert", Composto);
//recuperando o argumento do termo composto no vetor de termos "variavel"
PlCall("hello", variavel);
cout<<(char *)variavel[0];
```

OBS.: *chamadas ao Prolog serão abordadas na próxima seção.*

PlQuery

Representa a abertura e enumeração das soluções para uma consulta Prolog e encapsula as chamadas ao Prolog.

Construtores

PlQuery::PlQuery(const char *nome, const PlTermv &argumentos)

Cria uma consulta onde "nome" define o nome do predicado e "argumentos" o vetor de argumentos. A aridade é deduzida a partir do vetor de argumentos. O predicado está localizado no módulo Prolog do usuário.

Exemplo:

```
PlTermv vetor(3);
vetor[1]= 10;
vetor[2]= 25;
PlQuery q("compare",vetor); //fazendo a consulta
//"compare" determina a relação entre os termos, ou seja <, > ou =
q.next_solution(); //será explicado no final da seção
cout<<(char *)vetor[0];
```

int PlCall(const char *predicado, const PlTermv &argumentos)

Cria uma PlQuery dos argumentos, gera a primeira solução e destrói a solução. Retorna o resultado da solução ou uma exceção.

Exemplo:

```
//o código abaixo pertence ao segundo exemplo da seção PlCompound
//variavel é um PlTermv contendo argumentos e "hello" é o predicado
PlCall("hello", variavel);
```

Operador next

int PlQuery::next_solution( )

Fornece a próxima solução da consulta. Retorna 1 (*true*) se bem-sucedido e 0 (*false*) se não há mais soluções. Um exemplo as foi dado quando uma consulta foi feita usando PlQuery no início desta subseção.

PlFrame

Esta classe utilitária pode ser usada para descartar referências não utilizadas, bem como fazer "retrocesso de dados". A classe PlFrame fornece uma interface para descartar referências a termo não utilizadas, bem como retroceder unificações. Esta classe é muito útil se o programa de principal é definido em C ++ e chama Prolog várias vezes. Criar argumentos para uma consulta requer referências a termo e utilizar PlFrame é a única maneira de recuperá-las. Um uso típico para PlFrame é a definição de funções C ++ que chamam Prolog repetidamente.

Construtores

PlFrame::PlFrame( )

Cria uma instância dessa classe, e faz todas as referências a termo criadas depois dela serem válidas apenas no âmbito da presente instância.

Exemplo:

```
PlFrame frame;
{
      //todas as referências a termo criadas aqui são válidas
      //apenas no escopo da instância de PlFrame acima
}//a validade das unificações e referências acabam aqui
```

Destrutor

~ PlFrame( )

Libera todas as referências a termo criadas no escopo da instância de PlFrame.

Operador rewind

void PlFrame::rewind( )

Descarta todas as referências a termo e os dados criados na pilha global bem como desfaz todas as unificações, que estão no escopo da instância da PlFrame criada.

Exemplo:

```
PlFrame frame;
{
//pouco importa o que se deseja fazer, se o operador estiver dentro de um
//loop por exemplo, todas as referências a termo podem ser reutilizadas.
	while(/*condição*/){
		//qualquer código
		frame.rewind();
	}
}
```

PlEngine

Essa classe é usada em aplicações embarcadas, aplicações onde o controle principal é em C ++. Ela fornece a criação e a destruição do ambiente Prolog. A maioria assume que Prolog é a aplicação principal e C ++ é usado para adicionar funcionalidades ao Prolog, ou para acessar recursos externos ou por motivos de desempenho. Em algumas aplicações, é o contrário e queremos usar Prolog como um servidor de lógica. Por isso a classe PlEngine foi definida.

Apenas uma única instância desta classe pode existir em um processo. Quando usada em uma aplicação *multi-threading*, apenas uma *thread* por vez pode ter uma consulta em execução. As aplicações devem garantir isso usando técnicas de "*lock*" adequadas.

Construtores

PlEngine::PlEngine (*int* argc, *char* ** *argv*)

Inicializa o motor Prolog. A aplicação deve certificar-se de passar argv [0] da sua função principal, que é necessário na versão Unix para encontrar o executável em execução.

Exemplo:

```
//o vetor argv abaixo recebe uma dll, opções de inicialização e o caminho
// completo de um arquivo .pl
char* argv[] = {"swipl.dll","-q -s","C:/w/arquvo_prolog.pl", NULL};
//então um motor Prolog é criado usando os argumentos contidos em argv
```

```
PlEngine e(3,argv);
```

Destrutor

~ PlEngine ( )

Chamada a PL_cleanup( ) para destruir todos os dados criados pelo motor Prolog.

ANEXO B – BASE DE CONHECIMENTO

Módulo “act.pl”

```
% BC = Base de Conhecimento
% regra responsável por chamar a ação correspondente
rExecAct(Act,Arg):- % aqui é necessário receber parâmetros
      call(Act,Arg),!.

% diferencia ações que precisam de parâmetros das que não precisam
rExecAct(Act):- call(Act),!. % aqui não é necessário receber parâmetros

% responsável por "mover" o agente na BC
rActMove(X):-
      pAgenteEm(L),
      retractall(pUltimaSala(_)),
      assert(pUltimaSala(L)),
      retractall(pSalaVisitada(L)),
      assert(pSalaVisitada(L)),
      % verifica se é preciso mudar de sentido antes de seguir em frente
      rSentidoAgente(X),
      retractall(pAgenteEm(_)),
      assert(pAgenteEm(X)),
      pTempo(T),
      asserta(pAcao('moveu',T)),
      rUpdatePontos(-1),!.

% informa a BC que o agente encontrou o ouro e atualiza os pontos
rActOuro:-
      rUpdatePontos(1000),
      pTempo(T),
      asserta(pAcao('Pegou_Ouro',T)),!.
```

```
% muda o sentido do agente para o Oeste
rActOeste:-
      retract(pSentidoAgente(S)),
      retractall(pUltimoSentido(_)),
      assert(pUltimoSentido(S)),
      assert(pSentidoAgente(oeste)),
      pTempo(T),
      asserta(pAcao('Virou Oeste',T)),!.

% muda o sentido do agente para o Leste
rActLeste:-
      retract(pSentidoAgente(S)),
      retractall(pUltimoSentido(_)),
      assert(pUltimoSentido(S)),
      assert(pSentidoAgente(leste)),
      pTempo(T),
      asserta(pAcao('Virou Leste',T)),!.

% muda o sentido do agente para o Norte
rActNorte:-
      retract(pSentidoAgente(S)),
      retractall(pUltimoSentido(_)),
      assert(pUltimoSentido(S)),
      assert(pSentidoAgente(norte)),
      pTempo(T),
      asserta(pAcao('Virou Norte',T)),!.

% muda o sentido do agente para o Sul
rActSul:-
      retract(pSentidoAgente(S)),
      retractall(pUltimoSentido(_)),
      assert(pUltimoSentido(S)),
      assert(pSentidoAgente(sul)),
      pTempo(T),
      asserta(pAcao('Virou Sul',T)),!.

% sem ação...
nada.
```

Módulo “def.pl”

```
% os predicados precisam ser dinâmicos, pois mudam durante a execução
% "/2" indica a aridade do predicado, ou seja, o n° de argumentos
:- dynamic
([
      pAdjacente/2,           % relação de adjacência entre salas
      pAgenteVivo/0,          % determina se o agente está vivo ou não
      pDimensaoCaverna/1,     % tamanho da caverna
      pTempo/1,               % determina o tempo
      pAgenteEm/1,            % localização do agente
      pLimiteTempo/1,         % tempo máximo de execução do teste
      pSentidoAgente/1,       % leste, oeste, norte e sul
      pBrisa/1,               % percepção de brisa
      pFedor/1,               % percepção de fedor
      pBrilho/1,              % percepção de brilho
      pPossivelBuraco/1,      % onde o agente pensa existir um poço
      pPossivelWumpus/1,      % onde o agente especula que o Wumpus está
      pBuraco/1,              % buracos encontrados pelo agente
      pPontos/1,              % pontuação do agente
      pSalaSegura/1,          % indica que a sala é segura
      pSalaVisitada/1,        % informa que a sala foi visitada
      pAcao/2,                % indica quando e qual ação foi realizada
      pPercebe/3,             % percepção atual, seu local e o tempo
      pUltimoSentido/1,       % leste, oeste, norte e sul
      pUltimaSala/1,          % ultima sala que foi visitada
      pWumpusVivo/0,          % o Wumpus está vivo ou não
      pWumpusEm/1             % onde o agente encontrou o monstro
]).
```

Módulo “imprime.pl”

```
% tela inicial
rShowInit:-
      separador,
      rAgenteStatus(S),                  % agente vivo ou morto?
      format('Agente: ~w~n',[S]),        % imprime
      separador,
      pAgenteEm(L),                      % onde ele está?
      format('Agente em: ~w~n',[L]),     % imprime
      separador,
      pSentidoAgente(Se),                % qual o sentido do agente?
      format('Sentido: ~w~n',[Se]),      % imprime
      separador,
      pPontos(Pts),                      % pontuação
      format('Pontos: ~w~n',[Pts]),      % imprime
      separador2,
      !.
% separador é isso ---------------------------------------------------
separador:- format('~`-t ~80|',[]).
separador2:- format('~`-t ~80|~n',[]). % separador com quebra de linha
```

Módulo “infer.pl”

```
% BC = Base de Conhecimento
% percorre uma lista de salas e informa à BC que estas sala são seguras
rListaSegura([]).
rListaSegura([X|Y]):-
      retractall(pSalaSegura(X)),
      retractall(pPossivelBuraco(X)),
      retractall(pPossivelWumpus(X)),
      assert(pSalaSegura(X)),
      rListaSegura(Y).

% informa que as salas ao redor são seguras
rSalasSeguras:-
      pAgenteEm(L), % recupera o local do agente em L
      (
            retractall(pSalaSegura(L)),
            assert(pSalaSegura(L)),
            not(pBrisa(L)), % sem brisa em L
            not(pFedor(L)), % sem fedor em L
            setof(L, pAdjacente(L,Lista),_), % obtendo a vizinhança
            rListaSegura(Lista) % informa que vizinhança é segura

    ).
rSalasSeguras.

% informa à BC que em uma destas salas possivelmente está o Wumpus
rListaPWumpus([]).
rListaPWumpus([X|Y]):-
      retractall(pPossivelWumpus(X)),
      assert(pPossivelWumpus(X)),
      rListaPWumpus(Y).
```

```
% se o Wumpus foi achado, não faz sentido dizer que e ele
% pode estar em outras salas
rPossivelWumpus:- pWumpusEm(_), retractall(pPossivelWumpus(_)),!.

% informa que a vizinhança pode conter o monstro
rPossivelWumpus:-
      pAgenteEm(L), % local do agente em L
      retractall(pPossivelWumpus(L)), % L não pode conter o Wumpus
      pFedor(L), % percebe fedor em L
      setof(L, pAdjacente(L,Lista),_), % vizinhança de L
      exclude(pSalaSegura,Lista,Lista2), % exclui as salas seguras
      rListaPWumpus(Lista2). % as demais podem conter o monstro
rPossivelWumpus.

% informa que as salas na lista podem conter um poço
rListaPBuraco([]).
rListaPBuraco([X|Y]):-
      retractall(pPossivelBuraco(X)),
      assert(pPossivelBuraco(X)),
      rListaPBuraco(Y).

% informa que na vizinhança existe pelo menos um poço
rPossivelBuraco:-
      pAgenteEm(L), %local do agente
      retractall(pPossivelBuraco(L)),
      pBrisa(L), % percebe brisa
      setof(L, pAdjacente(L,Lista),_), % vizinhança de L
      exclude(pSalaSegura,Lista,Lista2), % eliminando as salas seguras
      exclude(pBuraco,Lista2,Lista3), % eliminado buracos encontrados
      exclude(pWumpusEm,Lista3, Lista4), % eliminando o Wumpus
      rListaPBuraco(Lista4). % informa quais são as salas perigosas
rPossivelBuraco.
```

```
% recupera a lista de salas que podem conter um poço e elimina as
% salas seguras, se restar apenas uma sala, então esta contem o buraco.
rAchaBuraco:-
      pAgenteEm(L), % local do agente
      pBrisa(L), % percebe brisa
      setof(L, pAdjacente(L,L1),_), % vizinhança de L
      exclude(pSalaSegura,L1,L2), % eliminando as salas seguras
      tamLista(L2,T2), % tamanho da lista
      (T2 == 1)
            ->
                  (
                        pertence(E,L2), % sala com o poço
                        retractall(pSalaSegura(E)),
                        retractall(pPossivelBuraco(E)),
                        retractall(pBuraco(E)),
                        assert(pBuraco(E))
                  ).
rAchaBuraco.

% se o Wumpus foi encontrado, não é preciso fazer nada.
rAchaWumpus:-pWumpusEm(_),!.

% recupera a lista de salas onde o Wumpus pode estar e elimina as salas
% seguras, se restar apenas uma sala, então é lá que o monstro está.
rAchaWumpus:-
      pAgenteEm(L),
      pFedor(L),
      setof(L,pAdjacente(L,L1),_),
      exclude(pSalaSegura,L1,L2),
      tamLista(L2,T2),
      (T2 == 1)
            ->
                  (
                        pertence(E,L2),
                        retractall(pSalaSegura(E)),
                        retractall(pPossivelWumpus(E)),
                        retractall(pWumpusEm(E)),
                        assert(pWumpusEm(E))
                  ).
rAchaWumpus.
```

Módulo “init.pl”

```
% inicia a base de conhecimento corretamente
rStart:-
% retractall em todos os predicados para garantir uma execução segura
% retractall retira da BC todas as afirmações usando predicados.
      retractall(pDimensaoCaverna(_)),
      retractall(pAdjacente(_,_)),
      retractall(pAgenteVivo),
      retractall(pTempo(_)),
      retractall(pAgenteEm(_)),
      retractall(pLimiteTempo(_)),
      retractall(pBrisa(_)),
      retractall(pFedor(_)),
      retractall(pBrilho(_)),
      retractall(pPossivelBuraco(_)),
      retractall(pPossivelWumpus(_)),
      retractall(pBuraco(_)),
      retractall(pPontos(_)),
      retractall(pPercebe(_,_,_)),
      retractall(pAcao(_)),
      retractall(pSalaSegura(_)),
      retractall(pUltimaSala(_)),
      retractall(pSentidoAgente(_)),
      retractall(pSalaVisitada(_)),
      retractall(pSalaSegura(_)),
      retractall(pWumpusVivo),
      retractall(pWumpusEm(_)),
      % asserções necessárias para iniciar a BC
      assert(pDimensaoCaverna(4)),
      assert(pAgenteVivo),
      assert(pTempo(0)),
      assert(pAgenteEm([1,1])),
      assert(pLimiteTempo(50)),
      assert(pSentidoAgente(leste)),
      assert(pSalaVisitada([1,1])),
      assert(pSalaSegura([1,1])),
      assert(pWumpusVivo),
      assert(pPontos(0)),
      assert(pUltimaSala([1,1])),
```

```
      % criando as adjacências entre as salas
      assert(pAdjacente([1,1],[[1,2],[2,1]])),
      assert(pAdjacente([1,2],[[1,1],[1,3],[2,2]])),
      assert(pAdjacente([1,3],[[1,2],[1,4],[2,3]])),
      assert(pAdjacente([1,4],[[1,3],[2,4]])),
      assert(pAdjacente([2,1],[[1,1],[2,2],[3,1]])),
      assert(pAdjacente([2,2],[[1,2],[2,1],[2,3],[3,2]])),
      assert(pAdjacente([2,3],[[1,3],[2,2],[2,4],[3,3]])),
      assert(pAdjacente([2,4],[[1,4],[2,3],[3,4]])),
      assert(pAdjacente([3,1],[[2,1],[3,2],[4,1]])),
      assert(pAdjacente([3,2],[[2,2],[3,1],[3,3],[4,2]])),
      assert(pAdjacente([3,3],[[2,3],[3,2],[3,4],[4,3]])),
      assert(pAdjacente([3,4],[[2,4],[3,3],[4,4]])),
      assert(pAdjacente([4,1],[[3,1],[4,2]])),
      assert(pAdjacente([4,2],[[3,2],[4,1],[4,3]])),
      assert(pAdjacente([4,3],[[3,3],[4,2],[4,4]])),
      assert(pAdjacente([4,4],[[3,4],[4,3]])),!.

% regra e inicia o programa de agente
rInit:-
      rStart,
      rShowInit,!.
```

Módulo “percep.pl”

```
% se o agente está morto não percebe nada
rPercebe(nada):-
      rAgenteStatus(S), % agente vivo?
      S == 'morto',
      pAgenteEm(X),
      pTempo(T),
      assert(pPercebe(S,X,T)),!.

% regra que coleta as informações e cria lista de percepções
rPercebe([F,B,O]):-
      call(rFedor,F), % percebe fedor?
      call(rBrisa,B), % percebe brisa?
      call(rBrilho,O), % percebe brilho?
      pAgenteEm(X),
      pTempo(T),
      % informando a percepção atual, na posição, X no tempo T
      assert(pPercebe([F,B,O],X,T)),!.

% verifica se o agente percebe fedor na posição atual
rFedor(fedor):-
      pAgenteEm(X),
      retractall(pFedor(X)),
      assert(pFedor(X)),!.
rFedor(nada). % caso contrário é retornado "nada"

% verifica se o agente percebe brisa na posição atual
rBrisa(brisa):-
      pAgenteEm(X),
      retractall(pBrisa(X)),
      assert(pBrisa(X)),!.
rBrisa(nada). % caso contrário é retornado "nada"
```

```
% verifica se o agente percebe brilho na posição atual
rBrilho(brilho):-
      pAgenteEm(X),
      % se ele já "pegou" o ouro, vai ignorar o quadrado amarelo
      not(pAcao('Pegou_Ouro',_))
            ->
                  (
                        retractall(pbrilho(X)),
                        assert(pbrilho(X))
                  ),!.
rBrilho(nada). % caso contrário é retornado "nada"

% o agente não percebe nada se está morto
rGetPercep(nada):-
      rAgenteStatus(S),
      S==morto,
      !.
% se está vivo
rGetPercep([F,B,O]):- % chama as regras que tratam das percepções
      call(rFedor,F), % regra de percepção do fedor
      call(rBrisa,B), % regra de percepção da brisa
      call(rBrilho,O), % regra de percepção do brilho
      pTempo(T), % recupera o tempo
      pAgenteEm(X), % local do agente
      %informa a pecepção no local e tempo
      assert(ptPercebe([F,B,O],X,T)),!.
```

Módulo “regras.pl”

```
:- use_module(library(random)). % necessario para aleatoriedde
% o teste termina quando:
rFim:-
      (
            not(pAgenteVivo),!; % o agente morreu
            pAcao('Pegou_Ouro',_),!; % o agente pegou ouro
            (
                  % ou o tempo limite foi atingido
                  pTempo(T),
                  pLimiteTempo(Tmax),
                  T = Tmax
            )
      )
            ->
                  (writeln('Fim.')),
      !.

% encontra uma sala não visitada na vizinhança
rSalaNaoVisitada(H):-
      pAgenteEm(X), % local agente
      pAdjacente(X,Lista),!, % vizinhança de X
      exclude(pSalaVisitada,Lista,L), % exclui as salas visitadas
      primeira(H,L),!. % pega a primeira

% encontra uma sala segura
rSalaSegura(H):-
      pAgenteEm(X), % local agente
      pAdjacente(X,Lista), % vizinhança de X
      filter(pSalaSegura,Lista,L), % cria uma lista só de salas seguras
      (L \= [[]]) % se a lista não for vazia
            ->
                  (
                        sort(L,L1), % ordena a lista
                        random_member(H,L1) % sorteia uma sala
                  ),!.
```

```
% retorna uma sala aleatória na vizinhança
rSalaAleatoria(S):-
      pAgenteEm(X), % local do agente
      pAdjacente(X,L), % vizinhança de X
      random_member(S,L),!. % sorteia uma sala

% verifica se é necessário mudar o sentido do agente
rSentidoAgente([H|T]):- % H = linha destino, T = coluna destino
      pAgenteEm([X|Y]), % coordenadas da localização do agente
      (
            (
                  (H < X) % se a linha atual é maior
                        ->
                              (rActSul); % ele vai para o sul
                  (H > X) % se a linha atual é menor
                        ->
                              (rActNorte) % ele vai para o norte
            ); % a linha atual e de destino são iguais
            (
                  (T < Y) % se a coluna atual é maior
                        ->
                              (rActOeste); % ele vai para o oeste
                  (T > Y) % se a coluna atual é menor
                        ->
                              (rActLeste) % ele vai para o leste
            )
      ),!.

% recupera o status do agente
rAgenteStatus(S):-
      (call(pAgenteVivo))
            ->
                  (S = vivo);
      S = morto.
```

```
% atualiza os pontos
rUpdatePontos(P):- % P pode ser negativo ou positivo
      pPontos(P1), % pontuação atual
      Pts is P1+P, % P negativo pontuação diminui, aumenta do contrário
      retractall(pPontos(_)), % removendo a pontuação atual
      assert(pPontos(Pts)). % atualizando a pontuação

% verifica se o agente morreu
rUpdateAgenteStatus:-
      pAgenteEm(L),
      % ele morre se estiver na sala do Wumpus ou de um poço
      (pWumpusEm(L); pBuraco(L))
            ->
                  (retractall(pAgenteVivo)).
rUpdateAgenteStatus.

% atualiza o status geral do teste
rUpdateStatus:-
      (
            rSalasSeguras, % informa salas seguras
            rPossivelBuraco, % informa salas que podem ter um poço
            rPossivelWumpus, % informa salas que podem abrigar o monstro
            rAchaBuraco, % tenta achar os poços
            rAchaWumpus, % tenta achar o Wumpus
            pTempo(E), % tempo atual
            E1 is E+1, % soma 1 ao tempo
            retractall(pTempo(E)),
            assert(pTempo(E1)) % atualiza o tempo
      ),!.
```

```
% reponsável pela execução do teste, pois chama as regras principais
rExecutar(Act, Percep):-
      separador,
      pTempo(E),
      format('Tempo: ~w || ',[E]),
      pPontos(Pts),
      format('~t Pontos: ~w~n',[Pts]),
      separador,
      pAgenteEm(L),
      pSentidoAgente(O),
      format('Agente em: ~w || ',[L]),
      format('Sentido Agente: ~w~n',[O]),
      % todas as linhas anteriores eram apenas para informação
      separador,
      rUpdateAgenteStatus, % atualiza o status do agente
      rGetPercep(Percep), % informa a percepção
      format('Percebe: ~w~n',[Percep]),
      separador,
      rUpdateStatus, % atualiza aspectos gerais do teste
      rEstrategia(Act,Percep), % consulta qual ação deve ser executada
      format('Estrategia: ~w~n',[Act]),
      separador,
      pSentidoAgente(Ss),
      pUltimoSentido(Us),
      format('Sentido Atual: ~w     ', [Ss]),
      format('Ultimo Sentido: ~w~n', [Us]),
      separador,
      (findall(S,pSalaSegura(S),LS))% lista sala seguras
            ->
                  (
                        sort(LS,LS1),
                        format('Salas Seguras: ~w~n',[LS1])
                  ),
      separador,
      (findall(P, pPossivelBuraco(P), LP))% lista possíveis poços
            ->
                  (
                        sort(LP,LP1),
                        format('Possiveis buracos: ~w~n',[LP1])
                  ),
      separador,
```

```
      (findall(P1, pBuraco(P1),Lp1)) % lista os poços encontrados
            ->
                  (
                        sort(Lp1,LP2),
                        format('Buracos em: ~w~n',[LP2])
                  ),
      separador,
      % salas onde o Wumpus pode estar
      (findall(W, pPossivelWumpus(W), LW))
            ->
                  (
                        sort(LW,Lw1),
                        format('Wumpus possivelmente em: ~w~n',[Lw1])
                  ),
      separador,
      (findall(W1,pWumpusEm(W1),LW1)) % onde o agente achou o Wumpus
            ->
                  (
                        sort(LW1,LW2),
                        format('Wumpus em: ~w~n',[LW2])
                  ),
      separador2,!.
```

Módulo “stratg.pl”

```
% chama a regra estratégia, passando a lista de percepções e obtém a
% ação correspondente
rCallEstg(Act, Percep):-
      rEstrategia(Act,Percep).

% ao receber nada como parâmetro não será executada nenhuma ação não
% importando a lista de percepções. serve para evitar erros
rCallEstg(nada,_):-
      rEstrategia(nada,_).

% se o agente está morto ele não percebe nada e nenhuma ação é executada
rEstrategia(morto,nada):-
      rAgenteStatus(S),
      (S == morto)
            ->
            (
                  write('Agente morto... Nenhuma '),
                  format('~s', [[97,231,227,111]]),
                  writeln(' a ser feita.'),
                  rUpdatePontos(-100)
            ),
      !.

% ao perceber brilho a ação correta é Pegar Ouro
rEstrategia('Pegar Ouro',[_,_,brilho]):-
      rExecAct(rActOuro),!.

% se está vivo e não percebe nada, é seguro avançar para a próxima sala
rEstrategia('Avante',[nada,nada,nada]):-
      rSalaNaoVisitada(S),  % de preferência uma sala não visitada
      (S \= [])
            ->
                  (rExecAct(rActMove,S));
      % se não houver sala não visitada então uma que seja segura
      rSalaSegura(S2),
      (S2 \= [])
            ->
                  rExecAct(rActMove,S2).
```

```
% se o Wumpus não foi encontrado a estratégia é ir para uma sala segura
rEstrategia('Sala Segura',[fedor,nada,nada]):-
      \+ pWumpusEm(_) % Wumpus não (\+) foi encontrado
            ->
                  rEstrategia('Sala Segura',[fedor,nada,nada]).

% se o Wumpus foi encontrado, então a vizinhança é segura, exceto a sala
% onde ele está
rEstrategia('Avante',[fedor,nada,nada]):-
            pWumpusEm(_),% Wumpus foi encontrado
            rSalaAleatoria(S),  % de preferência uma não visitada
            (S \= [], \+ pWumpusEm(S)) % onde não esteja o monstro
                  ->
                        (rExecAct(rActMove,S));
      % se não houver sala não visita, ir para uma que seja segura
            rSalaSegura(S1),
                        (S1 \= [])
                              ->
                                    rExecAct(rActMove,S1).

% regra genérica para evitar que o agente se arrisque e morra
rEstrategia('Sala Segura',[Fedor,Brisa,Brilho]):-
      (
            (Fedor == fedor; Brisa == brisa),
            Brilho == nada
      )
            ->
                  (
                        rSalaSegura(S),
                        (S \= [])
                              ->
                                    rExecAct(rActMove,S)
                  ),
      !.

% todas as estratégias anteriores falharam, não a nada a se fazer
rEstrategia('Sem Acao',[_,_,nada]):- rExecAct(nada),!.
```

Módulo “ult.pl”

```
% filtra os elementos para os quais o predicado P é verdadeiro
filter(_,[],[]).% caso base, fim da recursão
% P = predicado, [H|T] = lista, L = lista dos filtrados
filter(P, [H|T], L):-
      (
            (call(P, H)) ->  (L = [H|C]);
            L = C % a variável C nunca é instanciada.
      ),
      filter(P, T, C),!.

% exclui os elementos para o quais o Predicado P não é verdadeiro
exclude(_,[],[]).
exclude(P, [H|T], L):-
      (
            (not(call(P, H))) % perceba que a chamada está com negação
                  ->
                  (L = [H|C]); L = C % a variável C nunca é instanciada.
      ),
      exclude(P, T, C), !.

% adiciona um elemento no final de uma lista, nem sempre funciona
add([],L,L).
add(S,L,[S|L]).

% adiciona um elemento no final de uma lista, funciona bem
append([],L,L).
append([H|T],L2,[H|L3]):-append(T,L2,L3).

% verifica se o elemento S pertence a lista C
pertence(S,[S|_]).
pertence(S,[_|C]):- pertence(S,C).

% obtém o tamanho da lista
tamLista([],0).
tamLista([_|C],T):-tamLista(C,T2), T is T2+1.

% retorna o primeiro elemento da lista
primeira([],[]).
primeira(P,[H|_]):- P = H,!.
```

```
% une duas listas
merge([], L, L) :- !. % lista vazia unificada com uma lista é essa lista
merge(L, [], L) :- !. % uma lista unificada com lista vazia é ela mesma
merge([H1|T1], [H2|T2], [H|R]) :-
      (
            H1 @=< H2 % se a cabeça da lista 1 é menor...
                  ->  H = H1, % será a cabeça da nova lista
                        merge(T1, [H2|T2], R); % percorrendo o restante
            H = H2, % se não, a cabeça da nova lista é a da lista 2
            merge([H1|T1], T2, R) % percorrendo o restante
      ).
```

Módulo “w.pl”

```
% inicio sem mensagens
:-set_prolog_flag(verbose, silent).

% informa ao compilador os módulos que devem ser carregados
% nesse caso, estão todos no mesmo diretório, inclusive “w.pl”
:- ensure_loaded([act,def,infer,imprime,init,percep,regras,stratg,ult]).
```

ANEXO C – ARQUIVOS C++

Arquivo “mai.cpp”

```
#include <iostream>
#include <string>
#include <SWI-cpp.h>
#include "NXT++.h"
#include "functions.h"

using namespace std;

int main()
{
/* inicia o motor SWI-Prolog, o caminho do arquivo .pl deve estar correto. a dll correta também deve ser especificada, ela costuma mudar de nome de uma versão para outra do SWI-Prolog.
*/
      char* argv[] = {"swipl.dll","-q -s","C:/w3/w.pl", NULL};
      PlEngine e(3,argv);

      //Termo que vai receber uma string representando uma ação
      PlTerm Act;
      //Vetor de argumentos para construir a lista de percepções
      PlTermv vetor(3);
      //Lista de percepções
      PlTail Lista(vetor[0]);
      //Vetor de argumentos para o teste
      PlTermv Teste(Act, Lista);
      //String Prolog que contém o nome da regra principal
      PlString Consulta="rExecutar";
      //String C++ que é usada para construir a lista de percepções
      string percep="";
```

```
      //inicializa a base de conhecimento chamando a regra encarregada
      PlCall("rInit", 0);
      cin.get();
      system("cls");

      //cria um objeto que gerencia a conexão Bluetooth
      Comm::NXTComm comm;
      cout<<"\nConectando...\n";

      //tenta abrir a conexão
      if(NXT::OpenBT(&comm)){
            NXT::KeepAlive(&comm); //mantém ativa a conexão
            cout<<"\nConectado!\n";

//cria um objeto da classe PlFrame que é responsável por desfazer
//ligações de variáveis Prolog a termos Prolog
            PlFrame fr;
            {
                  //enquanto a regra rFim for falsa...
                  while(!(PlCall("rFim"))){
                        //elimina qualquer string contida em percep
                        percep.clear();
                        //coleta os dados e cria a lista de percepções
                        NXT_GetPercep(percep, comm);
                        //usa a string para construir a percepção
                        //c_str() converte uma string C++ em um char*
                        Lista = (PlCompound)percep.c_str();
                        //consulta a BC qual ação deve ser executada
                        PlCall(Consulta,Teste);
                        //ação retornada é transformada numa string
                        string Action = (string)Teste[0];
                        //executa a ação
                        //NXT_Act(Action, comm, 50);
                        //desfaz todas as ligações entre termos Prolog
                        fr.rewind();
                        cin.get();
                        system("cls");
                  }
            }
      }
```

```
      else
//não foi possível estabelecer a conexão entre o computador e o robô
            cout<<"A conexao falhou!\n";
      cin.get();
      return 0;
}
```

Arquivo “funcitions.h”

```
#ifndef FUNCTIONS
#define FUNCTIONS

/* FUNÇÃO PRINCIPAL */
//verifica a ação que deve ser executada e chama a função responsável
void NXT_Act(std::string Act, Comm::NXTComm &comm, const int &potencia);

/* FUNÇÕES MOTORAS */
/* potencia = 50% */
/* graus = 600  */
//aciona um dos motores
void NXT_AcionaMotor(Comm::NXTComm &comm, const int &potencia, const int
&motor, const int &graus);

//aciona os motores para que o robô avance
void NXT_Frente(Comm::NXTComm &comm, const int &potencia);

//ajusta a posição do robô após virar | atualmente não utilizada
void NXT_EmPosicao(Comm::NXTComm &comm, const int &potencia, const int
&graus);

//faz o robô dar meia volta
void NXT_MeiaVolta(Comm::NXTComm &comm, const int &potencia, const int
&motor1, const int &motor2, const int &graus);

//verifica se é preciso virar ou ainda se é preciso dar meia-volta
bool NXT_MudaSentido(Comm::NXTComm &comm, const int &potencia, const int
&graus);

//verifica a necessidade de mudar de sentido antes de seguir enfrente
void NXT_Sentido(PlTerm Sentido, PlTerm UltSentido, Comm::NXTComm &comm,
const int &potencia, const int &graus);

//desliga ambos os motores
void NXT_Stop(Comm::NXTComm &comm);

//faz o robô viarar para um lado
void NXT_Virar(Comm::NXTComm &comm, const int &potencia, const int
&motor, const int &graus);
```

```
/* FUNÇÕES PERCEPTIVAS */

// capta a cor através do sensor de cor
void NXT_GetColor(char *&fedor, char *&brisa, char *&brilho,
Comm::NXTComm &comm);

// monta a lista de percepção
void NXT_GetPercep(std::string &percep, Comm::NXTComm &comm);

/* FUNÇÕES AUXILIARES */
//cronômetro
void Cronometro(const float &segundos);

//verifica o sentido atual e o anterior para saber se é necessário virar
// ou dar meia-volta
void NXT_GetSentido(std::string &sentidoAtual, std::string &
sentidoAnterior);

//imprime uma lista Prolog | atualmente não utilizada
void PrintList(PlTermv vec);

#endif
```

Arquivo “functions.cpp”

```
#include <iostream>
#include "SWI-cpp.h"
#include "NXT++.h"
#include "functions.h"

/* FUNÇÃO PRINCIPAL */
//verifica a ação a ser tomada e chama a função correspondente
void NXT_Act(std::string Act, Comm::NXTComm &comm, const int &potencia){
      if((Act.compare("Sem Acao") != 0) && (Act.compare("Pegar Ouro") != 
0)){
                  NXT_MudaSentido(comm, potencia, 600);
                  NXT_Frente(comm, potencia);
      }
      else
            if(Act.compare("Sem Acao") == 0){
                  return;
            }
            else
                  if(Act.compare("Pegar Ouro") == 0){
                        return;
                  }
};

/* FUNÇÕES MOTORAS */
/* potencia = 45% graus = 600 */

//aciona um motor específico, faz parte da função NXT_Virar
void NXT_AcionaMotor(Comm::NXTComm &comm, const int &potencia, const int 
&motor, const int &graus){

      // zera os contadores de rotações dos motores A e C
      NXT::Motor::ResetRotationCount(&comm, OUT_A, false);
      NXT::Motor::ResetRotationCount(&comm, OUT_C, false);

      // aciona o motor que gira o número de graus com a potência 
desejada
      NXT::Motor::GoTo(&comm, motor, potencia, graus, true);
};
```

```
// move o robô para frente de acordo com a potência especificada
void NXT_Frente(Comm::NXTComm &comm, const int &potencia){

      // zera os contadores de rotações dos motores A e C
      NXT::Motor::ResetRotationCount(&comm, OUT_A, false);
      NXT::Motor::ResetRotationCount(&comm, OUT_C, false);

      //aciona os motores até atingirem os graus,
      {
            NXT::Motor::SetForward(&comm, OUT_A, potencia);
            NXT::Motor::SetForward(&comm, OUT_C, potencia);
      }
      Cronometro(2); // conta o tempo até parar
      NXT_Stop(comm); // desliga os motores
};

//ajusta a posição do robô após AcionaMotor | atuamlente não utilizada
void NXT_EmPosicao(Comm::NXTComm &comm, const int &potencia, const int
&graus){
      int graus1, graus2, ajuste;
      //obtendo a rotação dos motores e graus
      graus1= NXT::Motor::GetRotationCount(&comm, OUT_A);
      graus2= NXT::Motor::GetRotationCount(&comm, OUT_C);
      if(graus1 != graus2){ //se são diferentes é preciso ajustar
            if(graus1 < graus2){
            ajuste = graus1-graus2;
            //aciona o motor C para ajustar a posição
            NXT::Motor::GoTo(&comm, OUT_C, potencia, ajuste, true);
      }
            else{
            ajuste = graus2-graus1;
            //aciona o motor A para ajustar a posição
            NXT::Motor::GoTo(&comm, OUT_A, potencia, ajuste, true);
            }
            // zera os contadores de rotações dos motores A e C
            NXT::Motor::ResetRotationCount(&comm, OUT_A, false);
            NXT::Motor::ResetRotationCount(&comm, OUT_C, false);
```

```
            /* aciona os motores que giram o número de graus desejado
            com a potência especificada e param logo em seguida */
            NXT::Motor::GoTo(&comm, OUT_A, potencia, graus, true);
            NXT::Motor::GoTo(&comm, OUT_C, potencia, graus, true);
      }
      else{
            return;
      }
};

// faz o robô dar meia volta
void NXT_MeiaVolta(Comm::NXTComm &comm, const int &potencia, const int
&motor1, const int &motor2, const int &graus/*positivo*/){

// troca o sinal de "graus" e armazena em graus2 que deve ser positivo
      int graus2 = graus;
      graus2 *= -1;
//ajuste de 03 graus necessário por conta de imprecisão
      graus2 -= 3;
      // zera os contadores de rotações dos motores A e C
      NXT::Motor::ResetRotationCount(&comm, OUT_A, false);
      NXT::Motor::ResetRotationCount(&comm, OUT_C, false);

/* chama NXT_AcionaMotor 2x uma passando graus negativos, motor vai para
trás, outra passa graus positivos vai para frente
*/
      //virando
      NXT_AcionaMotor(comm, potencia, motor1, graus);
      NXT_AcionaMotor(comm, potencia, motor2, graus2/*negativo*/);
      NXT_Frente(comm, potencia); //ajusta a posição no quadrado
};
```

```
//verifica se é necessário virar o robô antes de seguir em frente
bool NXT_MudaSentido(Comm::NXTComm &comm, const int &potencia, const int
&graus){
      //OUT_A = motor direito
      //OUT_C = motor esquero
//sentido do agente antes da decisão (sentido físico)
      std::string sentidoAnterior;
//sentido do agente após a decisão (sentido na base de conhecimento)
      std::string sentidoAtual;
//recupera o sentido atual (BC) e o sentido anterior (sentido real)
      NXT_GetSentido(sentidoAtual, sentidoAnterior);
//se os sentidos são diferentes, é preciso descobrir qual direção o robô
// deve tomar
      if(sentidoAtual.compare(sentidoAnterior) != 0){
            if(sentidoAtual == "leste"){
                  if(sentidoAnterior == "oeste")
                  NXT_MeiaVolta(comm, potencia, OUT_A, OUT_C, graus);
//NXT_MeiaVolta é chamada quando se trata de sentidos opostos
                  else
                        if(sentidoAnterior == "norte"){
                              NXT_Virar(comm, potencia, OUT_A, -graus);
//NXT_Virar é chamada quando o robô deve simplesmente virar para um lado
                        }
                        else
                              if(sentidoAnterior == "sul"){
                              NXT_Virar(comm, potencia, OUT_C, -graus);
                              }
            }
            else
                  if(sentidoAtual == "oeste"){
                        if(sentidoAnterior == "leste")
                  NXT_MeiaVolta(comm, potencia, OUT_A, OUT_C, graus);
                        else
                              if(sentidoAnterior == "norte"){
                              NXT_Virar(comm, potencia, OUT_C, -graus);
                              }
                              else
                                    if(sentidoAnterior == "sul"){
                              NXT_Virar(comm, potencia, OUT_A, -graus);
                                    }
                  }
```

```
                  else
                        if(sentidoAtual == "norte"){
                              if(sentidoAnterior == "sul")
                  NXT_MeiaVolta(comm, potencia, OUT_A, OUT_C, graus);
                              else
                                    if(sentidoAnterior == "oeste"){
                              NXT_Virar(comm, potencia, OUT_A, -graus);
                                    }
                                    else
                                          if(sentidoAnterior == "leste"){
                              NXT_Virar(comm, potencia, OUT_C, -graus);
                                          }
                        }
                        else
                              if(sentidoAtual == "sul"){
                                    if(sentidoAnterior == "norte")
                  NXT_MeiaVolta(comm, potencia, OUT_A, OUT_C, graus);
                                    else
                                          if(sentidoAnterior == "oeste"){
                              NXT_Virar(comm, potencia, OUT_C, -graus);
                                          }
                                          else
                                    if(sentidoAnterior == "leste"){
                              NXT_Virar(comm, potencia, OUT_A, -graus);
                                                }
                              }
            return true; //retorna true se mudou de sentido
      }
      else{
            return false; //retorna false caso contrário
      }
};

//desliga os motores
void NXT_Stop(Comm::NXTComm &comm){
      NXT::Motor::Stop(&comm, OUT_A, true);
      NXT::Motor::Stop(&comm, OUT_C, true);
};
```

```
//faz o robô virar para um lado
void NXT_Virar(Comm::NXTComm &comm, const int &potencia, const int
&motor, const int &graus){
      NXT_AcionaMotor(comm, potencia, motor, graus);
      NXT_Frente(comm, potencia/2);
      Cronometro(1);
};

/* FUNÇÕES PERCEPTIVAS */

//coleta os dados através do sensor de cor e retorna uma cadeia de
// caracteres para cada percepção
void NXT_GetColor(char *&fedor, char *&brisa, char *&brilho,
Comm::NXTComm &comm){
      int cor=0;
      NXT::Sensor::SetColor(&comm, IN_3,'F');
      cor= NXT::Sensor::GetValue(&comm, IN_3);

            if(cor==2){
                  fedor="nada,";
                        brisa="brisa,";
                        brilho="nada";
            }
            else
                  if (cor== 3){
                        fedor="fedor,";
                        brisa="nada,";
                        brilho="nada";
                  }
                  else
                        if(cor==4){
                              fedor="nada,";
                              brisa="nada,";
                              brilho="brilho";
                        }
                        else
                              if(cor==5){
                                    fedor="";
                                    brisa="morto";
                                    brilho="";
                              }
```

```
                                else{
                                      fedor="nada,";
                                      brisa="nada,";
                                      brilho="nada";
                                }
        NXT::Sensor::SetColorOff(&comm, IN_3);
};

//cria a lista de percepções como uma cadeia de caracteres
void NXT_GetPercep(std::string &percep, Comm::NXTComm &comm){

      char *fedor;
      char *brisa;
      char *brilho;
      //coleta de dados
      NXT_GetColor(fedor,brisa,brilho,comm);
      //compondo a cadeia de caracteres
      percep= "[";
      percep+= fedor;
      percep+= brisa;
      percep+= brilho;
      percep+= "]";
};

/* FUNÇÕES AUXILIARES */

//conta o tempo para parar os motores
void Cronometro(const float &segundos){
      // tempo1 e tempo2 recebem a hora do sistema
      clock_t tempo1= clock(), tempo2= tempo1;
      // Determina o tempo do cronômetro
      clock_t tempo_final= segundos * CLOCKS_PER_SEC;
//a diferença entre tempo2 e tempo1 fica maior aproximando-se de
// tempo_final
      while((tempo2-tempo1) < tempo_final){
// tempo2 recebe o horário do sistema ficando maior com o tempo
            tempo2= clock();
      }
};
```

```
//verifica o sentido atual e o anterior para saber se é necessário virar
// ou dar meia-volta
void      NXT_GetSentido(std::string      &sentidoAtual,      std::string
&sentidoAnterior){
      PlTerm sentido, ultimoSentido;
      PlFrame fr;
      {
            PlCall("pSentidoAgente", sentido);
            PlCall("pUltimoSentido", ultimoSentido);
            sentidoAtual = (char*)sentido;
            sentidoAnterior = (char*)ultimoSentido;
            fr.rewind();
      }
};

//imprime uma lista Prolog em C++
void PrintList(PlTermv vec){
      PlTerm e;
      PlTail L(vec[0]);
      int i=0;

      std::cout<<"[";

      while(L.next(e)){
            std::cout<<(char *)e;
            if(i < L.arity())
                  std::cout<<", ";
      }
      std::cout<<"]";
};
```