O Mundo do Aspipó: Manual do libaspipo

por Ruben Carlo Benante

06/Março/2011

1- Introdução

Quando falamos de Inteligência Artificial (IA), um primeiro problema é a sua própria definição e campo de estudo. De modo geral, podemos separar os pesquisadores desta área em duas categorias: os que querem entender a inteligência humana, usando artifícios para tal; e os que querem criar alguma inteligência, qualquer tipo de inteligência, não importa se parecida com a humana ou não.

Em ambos os casos, os pesquisadores se valem dos computadores modernos para implementar e testar seus modelos. Mas a forma como os utilizam é completamente diferente para cada abordagem. Grosso modo, um pesquisador que tenta entender a inteligência humana, quer criar programas que modelem a mente, seus acertos e também seus erros, e possam auxiliar no tratamento de pessoas, ou possam prever atitudes e personalidades. O computador passa a ser um meio. Já os pesquisadores interessados em criar entidades inteligentes mesmo que sem semelhança humana, tendem a ver os problemas como processos lógicos a serem otimizados. A forma de solução dos problemas muda, quase nunca os humanos solucionam seus problemas de forma ótima (ou ideal). Uma rota pode ser escolhida em detrimento de outra por questões pessoais. Veja o caso do DeepBlue, o computador da IBM que venceu o ex-campeão mundial Garry Kasparov: este computador usava força bruta para calcular bilhares de lances em uma ordem sequencial dada pela posição das peças. Um ser humano que joga xadrez não pensa em todas as peças, nem mesmo seguem a ordem que estão no tabuleiro, e quem dirá, se conseguirem pensar em até 20 lances para frente, pode-se considerar um jogador profissional. Eles usam a intuição e escolhem poucos lances que lhes prendem a atenção, para solucionar sub-objetivos que são ameaças identificadas de pronto!

Uma forma de estudar IA é através do uso de agentes inteligentes. Agentes porque podem agir no meio, e suas ações são baseadas em percepções deste ambiente no qual estão inseridos. Então, até aqui, temos a sequência: percepção → processamento inteligente do agente → ação desenvolvida, e o ciclo se fecha incluindo a resposta do ambiente, que volta para a percepção.

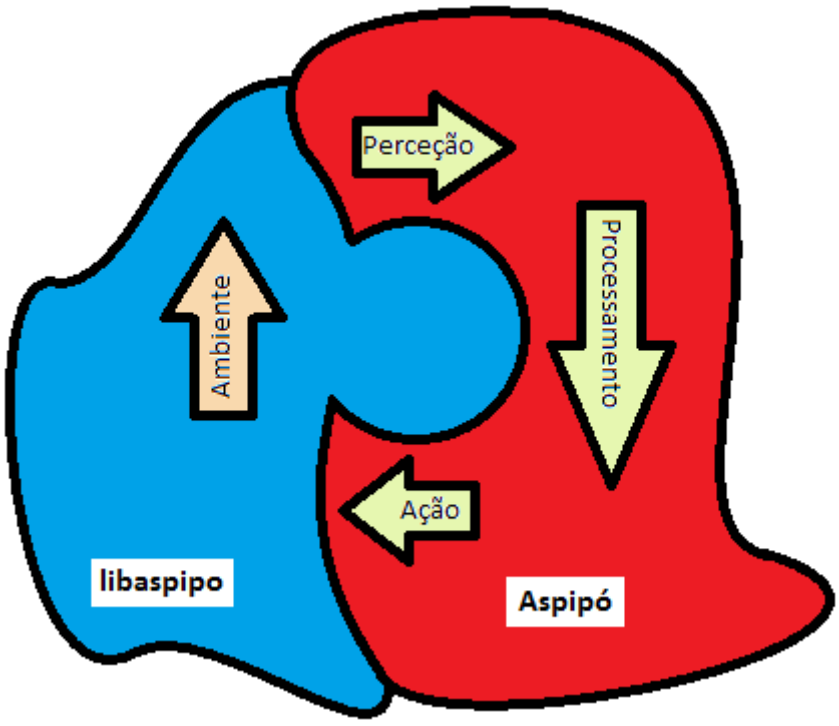

Figura 1: O ciclo de iteração entre o agente e o ambiente

Na Figura (1), o conjunto vermelho percepção-processamento-ação representa o agente. O ambiente, o conjunto azul dessa figura, é o simulador ou mundo real no qual ele está incluído, neste caso, realizado pelo **libaspipo**. Resta ainda determinar se este agente é “inteligente”.

Para conseguirmos, de modo restrito às simulações desta natureza aqui apresentada, considerar um agente inteligente, precisamos de um modo de como avaliarmos este agente. A isto denominamos **medida de desempenho** (MD).

Uma medida de desempenho deve ser apropriada para modelar matematicamente o objetivo do problema que o agente tentará resolver, de modo que o agente tenha como maximizar o seu ganho. Se o objetivo é claro, mas a medida de desempenho não é, ou se o modelo matemático da MD não corresponde ao objetivo, o que veremos serão agentes insólitos que não atingem o objetivo proposto, porém maximizam a medida de desempenho.

A MD é a equação que calcula o placar de um jogo. E os agentes inteligentes acabam por descobrir (ou são pré-programados) a agir da forma que lhes dê mais pontos, e se evite as punições. Portanto, é primordial modelar o objetivo que se deseja alcançar no mundo (ambiente) através de uma equação correta e precisa, que quando maximizada nos dê como resposta uma forma de resolver o problema proposto.

Então, um agente é dito inteligente, se consegue dentro dos seus limites de percepção/ação, uma pontuação de uma medida de desempenho ótima (ou boa), quando comparado com outros agentes, avaliados sob a ótica da mesma medida de desempenho.

A receita utilizada por um agente para maximizar a MD pode não agradar aos projetistas que procuram pela solução de um objetivo específico. A esta relação entre o objetivo desejado e o objetivo alcançado pelo agente ao maximizar uma MD dá-se o nome de **qualidade da MD**.

Quando a qualidade da MD é boa, um agente inteligente ao maximizá-la, nos dá como informação uma forma de resolvermos o problema que desejamos.

Vejamos na próxima seção uma definição formal do problema, e em seguida passaremos para explicações mais detalhadas do simulador **libaspipo**.

2- Definição do Problema

A biblioteca **libaspipo** implementa o **ambiente** de um simulador, para que possa ser utilizado por alunos e pesquisadores em IA para gerarem seus próprios agentes inteligentes e com isso terem um **benchmark** para o ambiente proposto. Este ambiente é inspirado no livro *Inteligência Artificial,* Russel & Norvig, 2.a ed., Campus, 2004, descrito na página 35.

O experimento descrito no livro é bastante simples, chamado "O mundo do Aspirador de Pó". Um agente inteligente apelidado carinhosamente de Aspipó se comunicará com este mundo, que está implementado na biblioteca **libaspipo**.

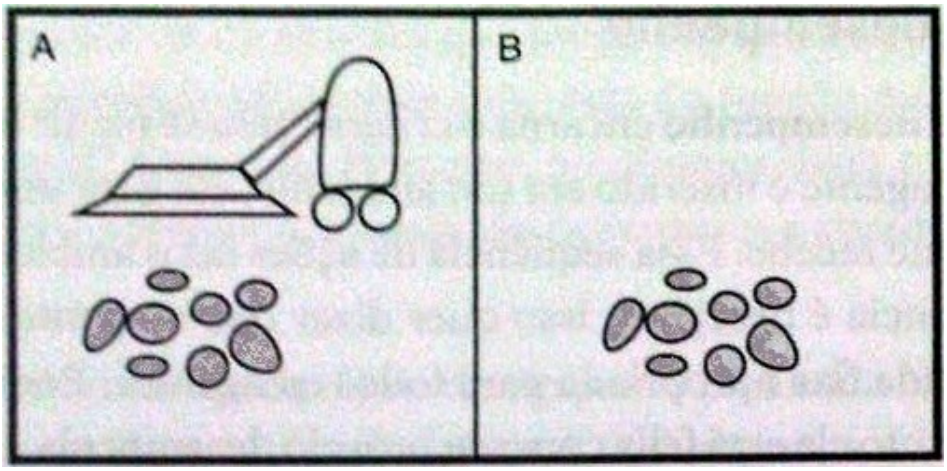

Figura 2: Mundo do Aspipó, um aspirador de pó com duas salas para cuidar (Russel & Norvig, p.35)

Uma descrição não formal do objetivo do agente é limpar (e manter limpas, caso voltem a se sujar) as salas A (ou sala zero) e B (ou sala um). O agente Aspipó tem um conjunto de recursos à sua disposição para realizar esta tarefa. Ele pode, por exemplo, se movimentar da esquerda para a direita e vice-versa, pode também aspirar uma determinada sala, e pode sentir através de seus sensores acoplados se a sala em que se encontra está limpa ou suja, e também pode perceber qual é a sua sala atual.

Mas antes de entendermos o agente em si, precisamos detalhar em que tipo de ambiente este agente atuará. Podemos categorizar os ambientes em relação a seis propriedades importantes inerentes aos mesmos.

Propriedades dos ambientes:

1- Completamente observável *versus* parcialmente observável.

O Mundo do Aspipó implementado na biblioteca **libaspipo** possui uma opção para tratar estes dois tipos de ambientes, que deve ser informada na inicialização. O mundo, conforme descrito na sua forma original, é parcialmente observável, e o agente só tem condições de saber se há sujeira na sala em que se encontra, através da leitura do seu sensor de sujeira. Em mundos parcialmente observáveis é importante manter um *modelo de mundo* para que informações colhidas *in loco* possam auxiliar na tomada de decisões quando o agente estiver em outro local (e portanto sem acesso sensorial à informação outrora colhida). Um ambiente completamente observável permite que o agente tenha acesso a todas as informações que são relevantes para a tomada de decisão de uma só vez. Uma informação é dita *relevante* se faz parte do cálculo da medida de desempenho. A grande vantagem é que o agente não precisa se lembrar (ter um modelo de mundo), pois pode a qualquer momento observar todos os estados do ambiente e tomar sua decisão. Por mais incrível e complexo que pareça, um jogo de xadrez é completamente observável, tudo o que o enxadrista precisa para tomar sua decisão está no tabuleiro diante dos seus olhos.

2- Determinístico *versus* Estocástico.

Um ambiente é dito determinístico se o seu próximo estado é determinado exclusivamente pela combinação do estado atual com a ação do agente, e nada mais interfere nesta configuração. Neste caso, o ambiente não atua, apenas o agente. Por exemplo, os jogos de *resta-um* ou de *palavras-cruzadas* são além de determinísticos, completamente observáveis. Esta combinação determinístico+observável permite que o agente não tenha que se preocupar com incertezas. Já um jogo de *paciência* é determinístico, porém parcialmente observável, o que aumenta a complexidade de sua

solução. Um ambiente estocástico se altera de modo indeterminado, após a ação do agente. Diz-se que o ambiente também é atuante. Por exemplo, ao dirigir seu carro, pode ocorrer um estouro de pneu, ou no caso do Mundo do Aspipó, podeme aparecer sujeiras ao acaso em algumas salas, ou o aspirador não possuir um mecanismo de sucção muito confiável. A biblioteca **libaspipo** implementa ambas opções, sendo, claro, necessário informar durante a inicialização, qual o tipo de ambiente que o agente atuará.

3- Episódico *versus* Sequencial.

Episódico é quando as ações atuais não influenciam as ações no futuro, o episódio atual é completamente independente do episódio seguinte. O Mundo do Aspipó é sequencial, pois uma ação errada em um momento (como esquecer de esvaziar o saco) influencia nas suas ações futuras (ficará sem espaço para aspirar ao encontrar nova sujeira e terá que reavalizar seus objetivos, voltando para esvaziar o saco). Um outro exemplo é o simples fato de andar para direita ou esquerda, pois ao se dirigir para um dos lados (digamos esquerda) o agente restringe sua ação de se dirigir para o outro lado na ação futura (só lhe resta a direita, no caso de um mundo com duas salas, ou então lhe restam menos salas à esquerda para mundos com mais salas).

4- Estático *versus* Dinâmico.

O ambiente é dito dinâmico se ele pode se alterar enquanto o agente está deliberando sobre qual ação realizar. O **libaspipo** implementa um mundo estático, baseado em turnos que sempre se alternam rigorosamente. Ora é a vez do agente atuar, ora é a vez do ambiente, e eles nunca se sobrepõe em tempo de realizar suas ações. Exemplos de ambientes dinâmicos são jogos de ação, em que o simples fato de ficar apenas parado pode lhe causar a morte!

5- Discreto *versus* Contínuo.

Num modelo contínuo existe a presença de continuidade em uma ou mais variáveis que descrevem o ambiente ou o agente, como o tempo, os estados de configuração do ambiente, os estados do agente, ações e sensores de percepção analógicos. O **libaspipo** implementa um ambiente discreto, com um número determinado de salas no ambiente, um número de ações e de percepções que podem ser realizadas pelo agente, e em turnos de tempo discretos.

6- Agente único *versus* Multiagente Competitivo *versus* Multiagente Cooperativo

O ambiente **libaspipo** é um ambiente de um único agente. Mas é possível imaginar a comunicação do ambiente com dois ou mais agentes e assim permitir um ambiente multiagente. Os ambientes multiagentes competitivos são aqueles em que os agentes competem pelos recursos do ambiente, tentando não só maximizar sua MD como também, se possível dentro de ações disponíveis, minimizar a MD do(s) agente(s) adversários. No caso do Mundo do Aspipó, apesar de lúdica a competição, tem muito mais sentido falarmos em ambiente cooperativo, em que ambos podem trocar informações sobre áreas que serão de sua responsabilidade, para que otimizem o processo de limpeza de todas as salas.

Uma vez detalhadas as propriedades dos ambientes, podemos começar a entender como o nosso agente trabalha neste ambiente.

Propriedades do Agente:

O agente ASPIPÓ pode ser descrito basicamente por um conjunto de ações que estão disponíveis, e de sensores que pode lhe trazer informação do ambiente. Como antes de agir, é bom pensar, vamos primeiramente descrever os sensores:

Sensores:

- Sensor de Sujeira: permite ao agente detectar a presença de sujeira na sala em que se encontra.
- Sensor de Posição: indica ao agente o número da sala em que se encontra.
- Sensor de Sala de Descarga: indica ao agente se a sala em que ele se encontra possui ou não uma dispensa para esvaziar o saco do aspirador de pó.
- Sensor de Chamado: retorna o número de alguma sala suja que está chamando pelo auxílio do Aspipó.
- Sensor de Terremotos: quando ocorre um terremoto, este sensor automaticamente indica o abalo sísmico.

Como se pode perceber, é com a informação vinda dos sensores que o agente deve seguir sua lógica. Usa-se o termo **percepção** para descrever as entradas que o agente recebe do ambiente. Agentes simples podem atuar baseados apenas na sua última percepção, mas na prática os agentes inteligentes complexos baseiam-se em toda a **sequencia de percepções** que compõe a história do agente. O agente é descrito abstratamente por uma **função agente** que implementa de modo concreto um **programa agente**.

Além das percepções do mundo, o agente deve ser capaz de atuar para modificar o mundo e conseguir atingir seus objetivos (conseguir deixar o mundo em uma configuração desejável). As ações que o agente Aspipó pode realizar são:

Ações:

- Mover para esquerda: faz com que o agente mova para a sala da esquerda (de número inferior).
- Mover para direita: move o agente para a sala da direita (de número superior).
- Aspirar: aciona o mecanismo de sucção para que a sujeira da sala seja aspirada e a sala limpa.
- Assoprar: aciona o mecanismo de sucção de modo reverso, eliminando toda a sujeira contida no compartimento do agente (e claro, sujando a sala).
- Passar a vez: quando o agente acredita (por seu modelo de mundo, ou por suas leituras de sensores) que o mundo como está está muito bem e é melhor não mexer que estraga, então ele pode passar a vez.

Algumas ações parecem não ter sentido sem uma explicação mais detalhada das opções que o agente pode se defrontar. Mas cada uma terá uma função importante em cada caso, apenas entenda-as para usá-las quando apropriado. É apropriado agora definirmos um problema formalmente, para que possamos entender melhor os conceitos aqui apresentados, e resolvê-lo utilizando a biblioteca **libaspipo**.

Problema Formal:

Um problema para ser **bem definido** precisa especificar quatro propriedades básicas:

1. **Estado inicial:** é o estado em que o agente e o ambiente se encontram no começo da simulação.
2. **Descrição das ações possíveis do agente:** uma descrição precisa de cada ação e sua consequência no ambiente.
3. **Teste objetivo:** determina se um dado estado do ambiente é considerado um objetivo. Por exemplo, numa rota de carro num mapa, o estado objetivo é a cidade destino. Num jogo de xadrez, o estado objetivo é o xeque-mate. O teste objetivo para determinar se uma dada posição

é ou não um xeque-mate requer muitos cálculos sobre posições de peças, seus ataques e possíveis movimentos. No agente Aspipó, o seu objetivo é conseguir uma configuração do ambiente com todas as salas limpas. Isso pode ser fácil, se o ambiente for determinístico, mas pode requerer constante atenção do agente caso o ambiente seja estocástico.

4. **Função de custo:** esta função permite calcular o peso de cada ação. Os pesos são usados para calcular a medida de desempenho, e portanto a função de custo deve refletir a MD. O custo de um caminho é a soma dos custos individuais de cada ação.

Além destas quatro propriedades, ainda temos que descrever o ambiente, conforme as propriedades dos ambientes vistas em seção anterior. Vamos começar a *supor* alguns fatos, para que tenhamos um exemplo *concreto*. Este será um problema básico e simples possível, porém sua definição será completa, para que outros problemas possam ser definidos da mesma maneira, alterando-se apenas as configurações de cada opção descrita. Veremos após este exemplo, um segundo exemplo ainda mais simples em relação à lógica, porém com mais acesso às variáveis da biblioteca **libaspipo**.

Conforme a Figura (1), podemos fazer a descrição do nosso ambiente pelas características que o criam, o seu **Estado Inicial**:

1. Quantidade de salas: fixa em 2
2. Número da sala menor e maior: aleatório entre zero e cem, conhecidas a priori pelo agente.
3. Posição inicial do agente: aleatória, desconhecida.
4. Definição das salas: Apenas 4% de chance de cada sala iniciar estando limpa, desconhecidas do agente.
5. Não há sala de descargas. Logo, a capacidade do saco do agente é infinita. Fato conhecido pelo agente

Uma vez definidas estas características, um ambiente é gerado. No caso da Figura (1), a posição inicial do agente é a sala zero, e ambas as salas estão sujas. A sala menor está numerada como zero, e a sala maior, um.

Continuando com a descrição de um **problema bem definido**, as **ações** que o agente pode realizar são como as já anteriormente descritas, divididas em duas partes: as ações **atuadoras** *esquerda*, *direita*, *aspirar*, e *passar a vez*; e as ações **sensoriais** *ler posição* e *ler sujeira*. O **teste objetivo** é manter as salas limpas, utilizando-se destas ações.

A **função custo** é dada por uma equação que leva em conta pesos para cada ação, e a quantidade de ações realizadas. A função custo completa implementada no ambiente **libaspipo** é:

```
    MD = qtd_andar*v_andar + qtd_ler*v_ler + qtd_aspirar*v_aspirar +
qtd_assoprar*v_assoprar + qtd_passarvez*v_passarvez + qtd_limpar*v_limpar +
qtd_descarregar*v_desc + qtd_bonus*v_bonus + qtd_tempolimpo*v_tempolimpo
```

O nosso agente, porém, só levará em consideração a quantidade de vezes que cada sala ficou limpa, por unidade de tempo, e a função custo é:

```
    MD = qtd_tempolimpo * v_tempolimpo
```

A variáveil iniciadas **qtd_tempolimpo** indicam a quantidade de bonificações, e as variável iniciadas por **v_tempolimpo** indica o peso, no nosso caso, 1.

Até aqui, criamos um problema bem definido, descrevendo seus quatro itens. Mas isto não basta para uma simulação completa. É preciso caracterizar o ambiente.

1. Completamente observável *versus* Parcialmente observável. Em mundos completamente observáveis, não precisamos de sensores, pois podemos colher a informação diretamente do mundo. O nosso primeiro exemplo será parcialmente observável. Nas configurações do **libaspipo** será o último nível, o nível 4 ou **OBS4**, completamente obscuro!

2. Determinístico *versus* Estocástico: nosso mundo será estocástico, e terá apenas uma característica estocástica: há uma probabilidade p_sujar = 0.05 (ou 5%) de cada sala se sujar sozinha a cada iteração. Sendo o nível 0, completamente determinístico, nós configuraremos o ambiente para o nível 1, que trata apenas de probabilidade de se sujar as salas. O último nível para esta configuração é o 127, totalmente estocástico. Os níveis são dados em combinações de propriedades que se deseja, criando os valores dos níveis em binário automaticamente através do operador binário |. Utilizaremos apenas a macro **DETSUJEIRA**. Mais detalhes sobre estas configurações serão descritas no apêndice.

Apesar do mundo ser parcialmente observável, isto se relaciona ao ambiente no decorrer da simulação. Ainda temos a chance de termos informação prévia, antes do início da simulação. Esta informação pode ser útil para um primeiro planejamento, mas logo se desatualiza. Chamamos esta informação ou conhecimento prévio do mundo de **conhecimento a priori**. O conhecimento à priori de nosso agente será das variáveis que indicam a quantidade de salas, qual o número da sala menor, e qual o número da sala maior, i.e. nível **APRIORI4**. Conhecimento total é dado escolhendo-se nível **APRIORI0**. O nível sem qualquer conhecimento a priori é o **APRIORI6.**

Toda esta informação nos permite finalmente inicializar a simulação com o comando:

```
 inicializar_ambiente(OBS4/*obs*/, DETSUJEIRA/*det*/, 2/*salas*/,
APRIORI4/*priori*/, DESC0/*descarga*/, &md1/*medida de desempenho*/,
NULL/*probabilidades*/);
```

Num ambiente com informação a priori, nosso agente da Figura (1) saberia que se encontra na sala zero, e poderia planejar seu próximo passo. No nosso ambiente configurado acima, ele não tem esta informação de antemão. Também continua sem acesso direto a essas informações durante a simulação, por termos configurado um ambiente **parcialmente observável**. No simulador, o número das salas é sempre aleatório. Então para saber se está na sala esquerda ou direita, o nosso agente, após consultar sua posição, deve confrontar esta informação com a informação a priori que tem sobre o número da sala menor e maior. O simulador rodará por **MAXACOES** (uma macro definida no valor de 1000 ações), ou até que o agente chame a função

```
finalizar_ambiente();
```

Durante estas 1000 ações, o agente deve manter suas duas salas limpas. Um algoritmo ótimo (para uma dada medida de desempenho) para resolver este problema, pode ser descrito como abaixo:

```
Inicializar_ambiente()
Enquanto não chega a MAXACOES:
         Se ler_sujeira(), então
                  aspirar()
         Se ler_posicao() == obs.menor_sala, então
                  direita()
         senão
                  esquerda
Fim-enquanto
Finalizar_ambiente()
```

Algoritmo (1): Solução para o problema do ASPIPO, 2 salas, parcialmente observável e estocástico

Programas agentes também podem ser representados por tabelas de *percepções* por *ações*. Para agentes simples, como este exemplo, a tabela pode ser representada graficamente. O algoritmo acima compreende a implementação da seguinte tabela de uma **função agente**:

Percepção de Posição	Percepção de Sujeira	Ação a realizar
Sala 0	Limpa	Direita
Sala 0	Suja	Aspirar
Sala 1	Limpa	Esquerda
Sala 1	Suja	Aspirar

Tabela (1): Tabela de percepções por ações para solução do problema 1.

Solução do primeiro problema:

O Programa (1) mostra a implementação completa de um código em linguagem C que se comunica com a biblioteca **libaspipo** e soluciona o problema descrito nesta seção de acordo com o Algoritmo (1) e a Tabela (1).

Para compilar este código, digite-o em um programa adequado para código (sugestões Kate, Kdevelop, Dev-CPP, ConText, Bloco de notas, entre outros) e salve-o em um arquivo com nome de **aspipo1.c**. Na mesma pasta em que foi salvo seu arquivo fonte, devem constar os arquivos **libaspipo.h** e **libaspipo.o**. Na linha de comando, estando na mesma pasta destes arquivos, digite:

- Para sistemas linux, se as dependências estiverem satisfeitas:

```
$ gcc aspipo1.c libaspipo.o -o aspipo1
```

- Para sistemas windows, caso tenha instalado o gcc.exe do Dev-CPP em pasta padrão:

```
$ c:\dev-cpp\bin\gcc.exe aspipo1.c libaspipo -o aspipo1.exe
```

Isto deve compilar e gerar o arquivo binário executável (aspipo1 ou aspipo1.exe). Execute o comando chamando-o da linha de comando. Se desejar analisar sua saída em um arquivo texto, use o redirecionamento do sistema operacional (ambos aceitam):

```
$ aspipo1.exe > saida1.txt
```

O cifrão ($) representa apenas o *prompt* de comando do *shell*, e não deve ser digitado. Assim poderá abrir a saída em seu editor de textos e a analisar com calma, além de permitir que seu programa rode muito mais rapido por não ter que fazer saída via monitor, que é demorada.

```
/* ASPIPO v.1 - aspirador de po inteligente                              */
/* Autor: Ruben Carlo Benante                                           */
/* email: rcb@beco.cc                                                   */
/* Copyright 2011. Licenca GNU/GPL: mantenha sempre o nome do autor     */
/* Template para exemplificar o problema numero 1 do manual.            */

#include <stdio.h>
#include "libaspipo.h"
int main(void)
{
  desempenho md1={0}; /*acessando a estrutura de cálculo de MD*/
  md1.v_tempolimpo=1; /*bonus por sala limpa por unidade de tempo*/

  /* Inicialização conforme definição formal do problema */
  inicializar_ambiente(OBS4, DETSUJEIRA, 2, APRIORI4, DESC0, FOLGA0, &md1, NULL);
  while(qtd_acoes()<=MAXACOES)
  {
    if(ler_sujeira()) /*se sujo*/
      aspirar(); /*aspiro*/
    if(ler_posicao()==obs.menor_sala) /*se estou na sala menor*/
      direita(); /*vou para direita*/
    else
      esquerda(); /*e vice-versa*/
  } /*repito ate acabarem as acoes*/
  finalizar_ambiente();
  pontos(); /*imprimir a pontuação final*/
}
```

Programa (1): Programa implementando a versão 1 do Aspipó.

Com este primeiro exemplo em mente, podemos analisar agora as possíveis variações deste problema de brinquedo (*toy-problem*) e tirarmos conclusões sobre as dificuldades apresentadas pelo ambiente para que tenhamos sucesso na criação de agentes realmente inteligentes.

3- Variações e suas complexidades

O Programa (1) não é o mais simples possível, dado o conjunto de configurações disponíveis no ambiente **libaspipo**. Para imaginarmos uma inicialização mais simples possível, teríamos:

- Um ambiente **completamente observável**. Neste caso não precisaríamos usar sensores. Bastaria acessar os valores das variáveis na estrutura observavel **obs**.
- Um ambiente **determinístico**. Assim, saberíamos que nossa atuação no mundo ocorreu exatamente como planejamos, e o estado do mundo após nossa ação não muda.
- **Quantidade de salas**: apenas duas salas, número fixo.
- Conhecimento **a priori** de toda informação do mundo.
- Sem necessidade de controlar o estoque no saco do aspirador, portanto sem **sala de descarga**.
- A simulação ocorre em no máximo MAXACOES iterações, finalizando. Portanto acertando a opção de **folga** como zero.
- A **medida de desempenho** considerará as seguintes pontuações/penalidades (as não citadas, serão zero):
  - andar perde 3 pontos
  - ler um sensor perde 1 ponto
  - aspirar ou assoprar, perde 10 pontos

- Conseguir uma aspirada bem sucedida (limpar a sala), ganha 100 pontos.
- Os demais valores são todos zero: passar a vez, descarregar em sala de descarga (já que nosso ambiente não conta com uma), sem bônus para salas limpas, e sem bônus por sala limpa por unidade de tempo.

- As **probabilidades** do mundo podem ser as *default*, pois não estamos em um mundo estocástico e não serão utilizadas. Colocamos **NULL**.

Com estes parâmetros acertados, temos o mundo mais fácil, claro, simples e cheio de informações que o agente pode encontrar. Ele não precisará ler os sensores, pois pode observar o mundo diretamente. Só possui duas salas, portanto pode aspirá-las na ordem que for mais rápida, a depender de onde iniciar sua posição. As salas não se sujarão novamente, portanto com algumas ações ele pode chamar a função **finalizar_ambiente()** e terminar, pois a missão estará cumprida. A inicialização é dada como:

```
inicializar_ambiente(OBS0, DETERMINISTICO, 2, APRIORI0, DESC0, FOLGA0, NULL, NULL)
```

Um programa completo **ótimo** precisaria apenas de 3 ações para realizar sua tarefa.

```
Se sala[minha posicao] esta suja
        aspirar()
fim-se
Se minha posicao == menor sala
        direita()
senao
        esquerda()
fim-se
Se sala[minha posicao] esta suja
        aspirar()
fim-se
```

Algoritmo (2): mundo mais simples possível

Em linguagem C, este programa completo seria:

```
#include <stdio.h>
#include "libaspipo.h"

int main(void)
{
  desempenho md1={0};

  md1.v_andar=-3;    /*penalidade por andar*/
  md1.v_ler=-1;      /*penalidade por ler algum sensor*/
  md1.v_aspirar=-10; /*penalidade por aspirar()*/
  md1.v_assoprar=-10;/*penalidade por assoprar()*/
  md1.v_passarvez=0; /*penalidade por passar a vez*/
  md1.v_limpar=100;  /*bonus por aspirar uma sala suja*/
  md1.v_desc=0;      /*bonus por assoprar na sala de descarga*/
  md1.v_bonus=0;     /*bonus por conseguir todas salas limpas simultaneamente*/
  md1.v_tempolimpo=0;/*bonus por sala limpa por unidade de tempo*/
inicializar_ambiente(OBS0, DETERMINISTICO, 2, APRIORI0, DESC0, FOLGA0, &md1,
  NULL);
  if(obs.sala[obs.mpos]==1) /*minha sala esta suja*/
    aspirar();
  if(obs.mpos==obs.menor_sala) /*comecei na sala esquerda*/
    direita();
  else
    esquerda();
  if(obs.sala[obs.mpos]==1) /*minha sala esta suja*/
    aspirar();
  finalizar_ambiente();
  pontos();
}
```

Programa (2): O programa mais simples possível, resolvido com 3 ações e 3 comparações

Note que não se faz uso das funções ler_sujeira() e ler_posicao(), pois as variáveis da estrutura observável obs cumprem seu papel de manter o mundo completamente observável, sem o filtro às vezes falhos de informações provindas de sensores. Salve como aspipo0.c e compile. Ao executar você verá uma saída do programa parecida com esta:

```
beco@simulador/unix-v1-02/libaspipo$ ./aspipo0
---------- agente (0):
inicializar_ambiente()
---------- ambiente:


---------- Configuracoes do ambiente:


--> Inicializacoes do ambiente:
qtdsala=2
MD: v_andar=-2, v_ler=-1, v_aspirar=-40, v_assoprar=-95, v_passarvez=0,
v_limpar=100, v_desc=100, v_bonus=100, v_tempolimpo=1
Probabilidades: p_sujar=0.050000, p_terremoto=0.010000, p_succao=0.250000,
p_movimento=0.100000, p_sensorial=0.100000
Menor sala = 30, Maior sala = 31
Sala de Descarga = -1, Capacidade do saco = -1
Posicao Inicial do ASPIPO: 31


--> Conhecimento a priori nivel (0-Tudo, 5-Nada): 0
```

```
posicao inicial do agente e quantidade de salas
limites das salas
condicoes de sujeira/limpeza de cada sala
capacidade do saco do aspirador
pesos da medida de desempenho e probabilidades dos eventos do ambiente

--> Deterministico x Estocastico nivel (0-Deterministico, 127-Estocastico): 0
Adicione o valor da opcao para obter o nivel:
 0 - Completamente Deterministico
 1 - Ambiente se suja com probabilidade p_sujar
 2 - O mecanismo de aspirar e assoprar falha com probabilidade p_succao
 4 - As rodas falham com probabilidade p_movimento
 8 - As leituras dos sensores retornam erradas com probabilidade p_sensorial
16 - Ocorrem terremotos com probabilidade p_terremoto, apos acoes, mas nao apos
leitura de sensores
32 - Ocorrem terremotos com probabilidade p_terremoto a qualquer tempo
64 - Capacidade do saco variavel

--> Observavel x Obscuro nivel (0-Observavel, 4-Obscuro): 0
Compartilhada na struct obs:
obs.terremoto (flag de terremoto)
ler_chamado() habilitada (retorna uma sala suja qualquer)
obs.mpos (posicao do agente)
obs.sala[i] (condicao da sala i de limpeza/sujeira)
obs.capasaco (capacidade do saco do agente)
obs.qtd_sala (quantidade de salas do ambiente)
obs.menor_sala e obs.maior_sala (limites do ambiente)
obs.descarga_sala (sala de descarga, se houver)
mostrar_pontos() habilitada (retorna a pontuacao corrente)
obs.v_andar, obs.v_ler, obs.v_aspirar, obs.v_assoprar, obs.v_passarvez,
obs.v_limpar, obs.v_desc, obs.v_bonus, obs.v_tempolimpo (valores dos pesos para
calculo da Medida de Desempenho)
obs.p_sujar, obs.p_terremoto, obs.p_succao, obs.p_movimento, obs.p_sensorial
(valores das probabilidades dos eventos do ambiente)

--> Sala de Descarga: nao
Funcao ler_descarga() desabilitada
Capacidade do saco infinita

--> Folga: nao
Acao maxima util: 1000
Programa ignora acoes entre 1001 e 1100
Programa aborta na acao 1101 se nao chamar finalizar_ambiente() no maximo na acao
1100
```

```
| 30| 31|
|   | A |
|***|***|
---------- agente (acao: 1):
aspirar()
---------- ambiente:
Aspirou e limpou a sala que estava suja!
Ja limpou 1 salas
Total de acao aspirar: 1
| 30| 31|
|   | A |
|***|___|
---------- agente (2):
esquerda()
---------- ambiente:
Andou para sala 30
Total de acao andar: 1
| 30| 31|
| A |   |
|***|___|
---------- agente (3):
aspirar()
---------- ambiente:
Aspirou e limpou a sala que estava suja!
Ja limpou 2 salas
Total de acao aspirar: 2
Ganhou bonus por manter todas salas limpas!
| 30| 31|
| A |   |
|___|___|
---------- agente (4):
finalizar_ambiente()
---------- ambiente:
Finalizada a simulacao.
Ganhou 10 X bonus por finalizar com todas salas limpas e agente na sala da
esquerda!
------------------------------------------------------------
pontos()
---------- ambiente:


Andadas                        =   1 *     -3 =      -3
Leituras                       =   0 *     -1 =       0
Aspiradas                      =   2 *    -10 =     -20
Assopradas                     =   0 *    -10 =       0
Passadas de vez                =   0 *      0 =       0
```

```
Limpezas                        =   2 *    100 =     200
Descargas                       =   0 *      0 =       0
Bonus Tudo Limpo                =  11 *      0 =       0
Ponto por sala limpa por tempo =   4 *      0 =       0

Seu agente fez 177 pontos
```

Tabela (2): Uma saída completa do programa mais simples

A saída do programa nos mostra passo-a-passo cada ação do agente e a reação do ambiente. No início, é despejada uma quantidade enorme de informação, explicitando todas as variáveis usadas para a inicialização e como elas alteraram a configuração do mundo. Vamos analisar cada parâmetro de inicialização e suas consequências no programa.

Logo em seguida, vem nosso agente inteligente, que resolveu o problema com 3 ações, sendo uma de andar e duas de aspirar! A sua primeira ação foi determinada pelo código:

```
  if(obs.sala[obs.mpos]==1) /*minha sala esta suja*/
    aspirar();
```

Note que o mundo, apesar de ter apenas duas salas, estas não eram numeradas 0 e 1, e sim 30 e 31. Mas isso não é problema para nosso agente onisciente, pois ele pode consultar a sua posição, que é o valor da variável **obs.mpos** e sabendo sua posição, pode acessar o vetor que indica as condições de cada sala, no caso **obs.sala[31]**, que foi onde ele começou. Com isso, descobriu que sua sala inicial estava suja, no que imediatamente aplicou uma aspirada. Como o mundo é **determinístico**, ele sabe que sua aspirada funcionou, e pode deixar essa sala para limpar a única restante.

Sua próxima decisão é saber se deve ir para esquerda ou direita. Para isso, ele precisa saber qual o valor da menor sala (ou análogo, da maior). Como ele tem acesso a esse conhecimento na estrutura **observavel**, ele compara sua posição com a da menor sala e se decide:

```
  if(obs.mpos==obs.menor_sala) /*comecei na sala esquerda*/
    direita();
  else
    esquerda();
```

Neste caso, as posições não coincidem, então ele deduz estar na "outra" sala, e portanto deve ir para esquerda. É a sua segunda ação. Estando na sala da esquerda, novamente ele não precisa de sensores para saber se deve ou não aplicar uma aspirada. Basta consultar a estrutura observável. Assim ele se decide a aspirar novamente, sua terceira e última ação, com o código idêntico ao da primeira decisão.

Não é necessário um laço (*while*) para que o agente volte a percorrer as salas e continuar a limpá-las. Trata-se de um ambiente determinístico, e portanto o agente sabe que as salas estão limpas e assim permanecerão até o fim dos tempos. Por isso, ele finaliza e pede uma estatística da sua pontuação:

```
  finalizar_ambiente();
  pontos();
```

A pontuação que se segue mostra números que precisam ser explicados. A tabela de 3 colunas tem na coluna da esquerda a quantidade de ações realizadas, em cada categoria, nas cinco primeiras linhas: *andadas*, *leituras*, *aspiradas*, *assopradas* e *passadas de vez*. Na coluna central, o valor de cada ação, que pode ser configurado passando uma estrutura **desempenho** para a função de **inicializar_ambiente()**. A terceira coluna tem a pontuação total de cada item. Os items *limpezas* e *descargas* se referem, respectivamente, ao tanto de aspiradas e assopradas realizadas com sucesso. Mesmo num mundo determinístico, um agente mal programado pode aspirar uma sala já limpa, caso em que não ganha o bônus, porém arca com o prejuízo da ação.

Nos valores determinados por nossa medida de desempenho, aspirar perde 10 pontos, mas

se aspirar corretamente, ganha-se 100 pontos, o que dá um lucro de 90 pontos para cada aspirada correta. No caso de assoprar, perde-se também 10 pontos, mas ganha-se 100 caso assopre na sala de descarga, num lucro também 90 pontos. Não é o caso desta configuração, pois não tem sala de descarga por nossa opção, portanto assoprar em qualquer sala sempre perderia os 10 pontos sem receber a bonificação de sucesso.

No início deste manual, enfatizamos a importância de termos medidas de desempenho que fossem compatíveis com os objetivos desejados pelo projetista. Veja que surpresa, este não é o caso aqui, e a medida de desempenho por nós determinada não é uma boa medida para uma configuração do ambiente como a que foi feita. Sendo o mundo determinístico, o agente não tem mais o que fazer após limpar suas duas salas, mas tem disponível 1000 ações para trabalhar. O que aconteceria se o agente resolvesse assoprar e aspirar novamente, repetindo o processo até esgotar suas ações? Uma assoprada lhe puniria com 10 pontos, mas ao aspirar, ganharia 100 pontos por uma limpeza bem sucedida! Estamos falando de um agente como este:

```
#include <stdio.h>
#include "libaspipo.h"

int main(void)
{
 desempenho md1={0};

 md1.v_andar=-3;    /*penalidade por andar*/
 md1.v_ler=-1;      /*penalidade por ler algum sensor*/
 md1.v_aspirar=-10; /*penalidade por aspirar()*/
 md1.v_assoprar=-10;/*penalidade por assoprar()*/
 md1.v_passarvez=0; /*penalidade por passar a vez*/
 md1.v_limpar=100;  /*bonus por aspirar uma sala suja*/
 md1.v_desc=0;      /*bonus por assoprar na sala de descarga*/
 md1.v_bonus=0;     /*bonus por conseguir todas salas limpas simultaneamente*/
 md1.v_tempolimpo=0;/*bonus por sala limpa por unidade de tempo*/

 inicializar_ambiente(OBS0, DETERMINISTICO, 2, APRIORI0, DESC0, FOLGA0, &md1,
 NULL);

 while(qtd_acoes()<=MAXACOES)
 {
   aspirar();
   assoprar();
 }

 finalizar_ambiente();
 pontos();
}
```

Programa (3): uma solução insólita para ganhar pontos burlando as regras do jogo

Este agente é uma aberração em se tratando de inteligência, mas como ele consegue estes resultados e tem um desempenho melhor que nosso agente inteligente do Programa (2)? São duas lições que devemos tirar destas situações: primeiro, é necessário validar a medida de desempenho para saber se ela representa matematicamente o objetivo que temos em mente. Segundo, nem sempre (ou seria quase nunca?) a mesma medida de desempenho pode ser usada para comparar programas configurados em ambientes diferentes.

```
pontos()
---------- ambiente:

Andadas                          =   0 *     -3 =       0
Leituras                         =   0 *     -1 =       0
Aspiradas                        = 500 *    -10 =   -5000
Assopradas                       = 500 *    -10 =   -5000
Passadas de vez                  =   0 *      0 =       0
Limpezas                         = 500 *    100 =   50000
Descargas                        =   0 *      0 =       0
Bonus Tudo Limpo                 =   0 *      0 =       0
Ponto por sala limpa por tempo = 500 *      0 =       0

Seu agente fez 40000 pontos
```

Tabela (3): pontuação do burlador, o Programa (4)

Estes dois programas (2) e (3) ao competir no ambiente estabelecido mostram que o que de fato ocorreu é que o programa (3) não burlou as regras do jogo, só soube melhor aproveitá-las, e não é culpa dele se o criador do jogo criou uma brecha. Mas então, como evitar essas brechas? Criar medidas de desempenho que quantifiquem com fidelidade um objetivo abstrato é quase tão complicado quanto criar a própria solução para o problema. E de fato, uma equação que modele um problema descreve sua solução. Fica como exercício descobrir uma medida de desempenho válida que coloque estes dois programas na sua devida ordem de coerência com o objetivo desejado.

Após estes exemplos vamos discutir item por item cada opção de configuração do ambiente **libaspipo**. Seguiremos os oito parâmetros da função **inicializar_ambiente()**, para depois passarmos por outros detalhes do ambiente.

```
inicializar_ambiente(Um, Dois, Três, Quatro, Cinco, Seis, Sete, Oito);
```

1- Nível de Observabilidade:

O primeiro parâmetro define se o ambiente é completamente observável, ou parcialmente observável, e se parcialmente, quais as variáveis que estão disponíveis para o agente. A configuração deste parâmetro é incremental, ou seja, o valor mais completo (número mais baixo) permite alguma propriedade além de todas já permitidas pelos valores mais altos. Este parâmetro pode receber os seguintes valores:

- **OBS4**: Nada pode ser observado a não ser por leitura de sensores.
- **OBS3**: Fica disponível a flag para terremotos **obs.terremoto**
- **OBS2**: A função **ler_chamado()** fica habilitada, além do permitido por OBS3.
- **OBS1**: É possível consultar a posição do agente em **obs.mpos**, a condição de limpeza de cada sala, no vetor **obs.sala[i]**, a capacidade de armazenamento do saco em **obs.capasaco** (quando habilitada a sala de descarga), a quantidade de salas do ambiente em **obs.qtd_sala**, além do que já está permitido por OBS2 e OBS3.
- OBS0: Além de tudo o que já foi permitido acima, também pode-se acessar as variáveis de peso da equação de medida de desempenho **obs.v_andar**, **obs.v_ler**, **obs.v_aspirar**, **obs.v_assoprar**, **obs.v_passarvez**, **obs.v_limpar**, **obs.v_desc**, **obs.v_bonus** e **v_tempolimpo**, bem como as variáveis que determinam os eventos estocásticos **obs.p_sujar**, **obs.p_terremoto**, **obs.p_succao**, **obs.p_movimento** e **obs.p_sensorial**.

obs.terremoto: Quando habilitados os terremotos (veja mais sobre eles no próximo parâmetro), o agente deve após cada ação sua, verificar se a variável de flag está com valor 1, caso em que indica que ocorreu um terremoto. Se não verificar, na sua próxima ação esta variável terá valor 0 (é dita ter valor volátil). Caso se queira lembrar da ocorrência de um terremoto em um ponto específico do código, é preciso salvar o valor desta variável em uma variável local.

ler_chamado(): Esta função retorna o número de uma sala qualquer aleatória, que esteja

suja. Pode ser usada para guiar o agente à distância, dando-lhe um sub-objetivo. Caso nenhuma sala esteja suja, ela retorna -1. Como é uma leitura sensorial, está sujeita a ruídos que são determinados pela probabilidade **p_sensorial**. Se retornar um valor com ruído, poderá retornar uma sala limpa, ou mesmo a sala de descarga!

obs.mpos: diz a posição real do agente. Não é leitura de sensor, portanto está sempre representando a realidade do mundo.

obs.capasaco: Se habilitada a opção de ter uma sala de descarga entre as salas do ambiente, esta variável, se disponível, dá o número de limpezas que o agente consegue aspirar antes de ter que assoprar seu saco na descarga. Este número pode ser estocástico, ou fixo no valor da quantidade de salas menos 1, a depender das escolhas no parâmetro seguinte de determinismo.

obs.qtd_sala: contém a quantidade de salas do ambiente. Só é necessária sua consulta se optar por um número aleatório de salas no mundo.

As variáveis de peso da equação de medida de desempenho e de probabilidades para os eventos estocásticos já foram discutidas em momento anterior, nas equações apresentadas.

2- Nível de Determinismo:

O segundo parâmetro define se o ambiente é determinístico ou estocástico. Caso determinístico, as ações do agente são as únicas que modificam o mundo. Caso estocástico, é necessário especificar quais ruídos serão introduzidos no mundo. A especificação é feita através de máscaras binárias que podem ser combinadas individualmente com o conectivo de operação binária | (ou). As opções disponíveis somam valores de 0 (determinístico) a 127 (todas opções estocásticas selecionadas). Combinar a máscara DETERMINISMO com as outras estocásticas mantém apenas as estocásticas. Para se ter determinismo puro, deve-se usar sua máscara sozinha. As máscaras que podem ser combinadas são:

- **DETERMINISTICO**: nada acontece no mundo a não ser as ações do próprio agente.
- **DETSUJEIRA**: As salas se sujam ao acaso com probabilidade **p_suja**. O valor *default* é 5%
- **DETSUCCAO**: As ações de **aspirar()** e **assoprar()** estão sujeitas a falhas com probabilidade dada por **p_succao**. O valor *default* é de 25%. Caso ocorra um ruído, a ação não é executada.
- **DETMOVIMENTO**: As ações de movimento para **esquerda()** e **direita()** podem falhar com probabilidade **p_movimento**. O valor *default* é de 10%. Caso ocorra um ruído, o Aspipó não se move.
- **DETSENSORES**: Se introduzidos ruídos nos sensores, eles são regidos pela probabilidade dada em **p_sensorial** que é por *default* de 10%. Os sensores têm comportamentos diferenciados quando ocorre um ruído. Os sensores **ler_sujeira()** e **ler_descarga()** simplesmente invertem suas saídas, indicando sala limpa quando suja e vice-versa para o primeiro, e sala normal quando sala de descarga e vice-versa para o segundo. O sensor **ler_posicao()** retorna um valor alterado para +1 ou -1 do valor da posição real do Aspipó. O sensor **ler_chamado()** como já dito, retorna com ruído uma sala limpa ou a sala de descarga, ou -1 se não houver salas limpas nem de descarga.
- **DETTERREMOTOBASICO**: Os terremotos no modelo básico não ocorrem após leituras de sensores, somente ocorrem, com probabilidade **p_terremoto** apos acoes de movimento ou sucção. Quando um terremoto ocorre, ele muda a posição do agente para uma posição aleatória qualquer, e suja todas as salas. O modelo básico permite que o agente confira sua posição com uma leitura do sensor **ler_posicao()** sem que a própria leitura possa ocasionar outro terremoto e mudar novamente a posição do agente. A probabilidade *default* para esta variável é de 1%.
- **DETTERREMOTOTOTAL**: A variável de probabilidade para o modelo de terremoto total é a mesma do modelo básico. A diferença aqui é que os terremotos podem ocorrer a qualquer tempo, após qualquer tipo de ação do agente, mesmo que apenas a leitura de sua posição. Deste modo, o agente nunca é capaz de afirmar com certeza em que posição que está (a não ser, é claro, caso tenha acesso à variável **obs.mpos**). Caso o programador combine esta máscara com a anterior, a que fica atuando é a esta (terremoto total).
- **DETCAPACIDADE**: Sem esta opção estocástica capacidade de armazenamento do saco do agente é igual ao número de salas do ambiente menos um. Com esta máscara, a capacidade do

saco é inicializada com um valor aleatório no intervalo [teto(qtd_sala/3) , (qtd_sala-1)]. Uma pequena tabela explica este intervalo:

Qtd. salas	mín.	máx.
2	1	1
3	1	2
4	2	3
5	2	4
6	2	5
7	3	6
8	3	7
9	3	8
10	4	9

Tabela (4): Capacidade mínima e máxima do saco

3- Número de salas: Este parâmetro pode receber valores de 2 a 10, para se determinar de antemão qual a quantidade de salas do ambiente. Caso seja colocado um valor zero (ou fora do intervalo permitido), o número de salas será aleatório (entre 2 e 10).

4- Conhecimento a priori: É o conhecimento que o agente tem do mundo antes de iniciar a simulação. Este conhecimento estará disponível na estrutura **observavel obs** , da mesma forma que o indicado no parâmetro 1, sobre observabilidade. A diferença é as variáveis escolhidas no parâmetro 1 continuam sendo atualizadas durante a simulação. Já as variáveis escolhidas para conhecimento a priori estão congeladas. As variáveis de conhecimento imutável são informações importantes, mas as de valor mutável no tempo se perdem logo que a simulação inicia. As variáveis determinadas pelo parâmetro 1 continuam sendo atualizadas, independente do que for escolhido aqui. As opções deste parâmetro são cumulativas, iniciando do nível **APRIORI6** em que nada é conhecido, até o nível **APRIORI0**, em que toda a estrutura é conhecida. Estes níveis liberam, respectivamente, as variáveis:

- **APRIORI6**: Nenhum parâmetro conhecido a priori.
- **APRIORI5**: A quantidade de salas (**obs.qtd_sala**).
- **APRIORI4**: Além dos anteriores, os limites das salas (**obs.menor_sala** e **obs.maior_sala**).
- **APRIORI3**: Cumulativamente os anteriores e também a posição inicial do agente (**obs.mpos**).
- **APRIORI2**: Além das anteriores, o vetor que indica a condição das salas (**obs.sala[i]**).
- **APRIORI1**: Todos anteriores e a capacidade do saco (**obs.capasaco**).
- **APRIORI0**: Tudo, incluindo: a posição da sala de descarga (**obs.descarga_sala**), as variáveis de peso da medida de desempenho, e as variáveis de probabilidades dos eventos estocásticos.

5- Sala de descarga: Esta flag indica se haverá ou não sala de descarga no ambiente. Caso seja habilitada com a opção **DESC1**, ficam também habilitadas a capacidade do saco, e a função **ler_descarga()**. Caso seja desabilitada com a opção **DESC0**, o sensor **ler_descarga()** ficará quebrado, porém utilizá-lo está sujeito à penalidades previstas em MD, e a capacidade do saco se torna infinita.

6- Folga para terminar o serviço: As opções **FOLGA1** e **FOLGA0** habilitam e desabilitam, respectivamente, esta propriedade.

Quando habilitada, o agente terá 1000 (**MAXACOES**) ações para atuar no mundo, durante as quais o mundo reagirá de acordo com o estabelecido. Após estas **MAXACOES**, o agente terá até a iteração 1100 (**MAXACOESFINAL**) para operar em um mundo que não reage mais. Durante este período, o mundo deixa de se sujar (**p_sujar**) e não ocorrem mais terremotos (**p_terremoto**). Portanto, é um período apenas para uma limpeza final, e também para não pegar o agente desprevenido no meio de algum laço. É obrigação do agente chegar a quantidade de ações já realizadas pela chamada de **qtd_acoes()**. O agente deve chamar a função **finalizar_ambiente()** até no máximo a ação número **MAXACOESFINAL**, pois caso passe esse limite o ambiente será abortado e o programa terminado

com perda da pontuação.

Quando desabilitada por **FOLGA0**, o agente tem **MAXACOES** úteis no ambiente. Após este limite, o ambiente passa a **ignorar** completamente as ações do agente, esperando por até **MAXACOESFINAL** para que o agente chame **finalizar_ambiente()**. Se o ambiente não for finalizado neste período, o agente perderá pontos e o programa será abortado.

7- Estrutura **desempenho**: Este parâmetro espera um endereço para uma estrutura do tipo **desempenho**, que contém as variáveis com os pesos necessários para o cálculo da medida de desempenho. A estrutura é definida em **libaspipo.h** como:

```
typedef struct
{
  int v_andar, v_ler, v_aspirar, v_assoprar, v_passarvez;
  int v_limpar, v_desc, v_bonus, v_tempolimpo;
} desempenho;
extern desempenho md;
```

Veja o Programa (3) para um exemplo de uso.

8- Estrutura **probabilidade**: É definida em **libaspipo.h** como:

```
typedef struct
{
  float p_sujar, p_terremoto;
  float p_succao, p_movimento, p_sensorial;
} probabilidade;
extern probabilidade pr;
```

Estes valores são todos no intervalo de [0,1] e definem a frequência com que os eventos associados às variáveis ocorrem. Para alterar estes valores, utilize um código similar ao da estrutura **desempenho**. Veja o código exemplo do Programa (4) para criar uma nova medida de desempenho com novos parâmetros estocásticos ao final.

4- Exemplo de código para cálculo de medida de desempenho

```
/* ASPIPO - aspirador de pó inteligente                                  */
/* Autor: Ruben Carlo Benante                                            */
/* email: rcb@beco.cc                                                    */
/* Copyright 2011. Licença GNU/GPL: mantenha sempre o nome do autor      */


/*
   Template para exemplificar a configuração das variáveis de peso
para calculo da medida de desempenho, e das variáveis de probabilidade
que controlam os eventos do ambiente
   Os valores colocados nas struct desempenho e struct probabilidade
serão usados pelo ambiente para apresentar o resultado final.
   Caso se queira avaliar também outras medidas de desempenho, deve-se
usar a função gastos() que repassara as acoes do agente.
De posse das ações, calcula-se a equação de pontos desejada por uma somatória
ponderada simples.
*/

#include <stdio.h>
#include "libaspipo.h"

int main(void)
{
```

```
  desempenho md1={0};
  probabilidade pb={0};
  int i;

  md1.v_andar=-2;     /*penalidade por andar*/
  md1.v_ler=-1;       /*penalidade por ler algum sensor*/
  md1.v_aspirar=-40;  /*penalidade por aspirar()*/
  md1.v_assoprar=-95; /*penalidade por assoprar()*/
  md1.v_passarvez=0;  /*penalidade por passar a vez*/
  md1.v_limpar=100;   /*bônus por aspirar uma sala suja*/
  md1.v_desc=100;     /*bônus por assoprar na sala de descarga*/
  md1.v_bonus=100;    /*bônus por conseguir todas salas limpas simultaneamente*/
  md1.v_tempolimpo=1; /*bônus por sala limpa por unidade de tempo*/

  pb.p_sujar=0.05;    /*probabilidade de uma sala se sujar de 5% */
  pb.p_terremoto=0.01;/*probabilidade de ocorrer um terremoto de 1% */
  pb.p_succao=0.25;   /*probabilidade do mecanismo de sucção falhar de 25% */
  pb.p_movimento=0.2; /*probabilidade das rodas falharem de 20% */
  pb.p_sensorial=0.1; /*probabilidade dos sensores falharem de 10% */

  srand(time(NULL));

  inicializar_ambiente(OBS4, DETSUJEIRA|DETMOVIMENTO, 0/*QS*/, APRIORI0,
  DESC0, FOLGA0, &md1, &pb);

  printf("Agente diz: sou aleatorio!\n");
  while(1)
  {
    if(qtd_acoes()>MAXACOES) /*a partir daqui, o ambiente não se atualiza*/
      break;

    if(!(rand()%2)) /*50% das vezes, tenta aspirar*/
    {
      if(ler_sujeira())
        aspirar();
    }
    else
      if(!(rand()%2)) /*25% vai para direita, 25% vai para esquerda*/
        direita();
      else
        esquerda();
  }
  finalizar_ambiente();
  i=pontos();

  /*
  Aqui o agente já simulou e os pontos foram calculados com os pesos dados acima.
  Que por exemplificação, foram colocados os mesmos que são padrão do programa,
  podendo-se neste caso simplesmente usar NULL nos dois últimos parâmetros da
  função de inicialização.
  Para calcular novas medidas de desempenho para comparações, usa-se a função
  gastos() para obter as acoes do agente
  */

  int qtd_andar, qtd_ler, qtd_aspirar, qtd_assoprar, qtd_passarvez, qtd_limpar,
qtd_descarregar, qtd_bonus, qtd_tempolimpo;
  int qtd_sala, v_andar, v_ler, v_aspirar, v_assoprar, v_passarvez, v_limpar,
v_desc, v_bonus, v_tempolimpo;
  int medida2;

  gastos(&qtd_andar, &qtd_ler, &qtd_aspirar, &qtd_assoprar, &qtd_passarvez,
&qtd_limpar, &qtd_descarregar, &qtd_bonus, &qtd_tempolimpo);

  v_andar=10;
```

```
  v_ler=-1;
  v_aspirar=-5;
  v_assoprar=-25;
  v_passarvez=0;
  v_limpar=100;
  v_desc=25;
  v_bonus=1;
  v_tempolimpo=1;

  medida2=qtd_andar*v_andar + qtd_ler*v_ler + qtd_aspirar*v_aspirar +
qtd_assoprar*v_assoprar + qtd_passarvez*v_passarvez + qtd_limpar*v_limpar +
qtd_descarregar*v_desc + qtd_bonus*v_bonus + qtd_tempolimpo*v_tempolimpo;

  printf("Segunda medida de desempenho: %d\n", medida2);
}
```

Programa (4): Exemplo de cálculo de uma segunda medida de desempenho.

5- Resumo

Ações do Aspipó (não retorna valores úteis):

- **esquerda()** - move o Aspipó para esquerda. Pode falhar **p_movimento** vezes. Perde **v_andar** pontos.
- **direita()** - move o Aspipó para direita. Pode falhar **p_movimento** vezes. Perde **v_andar** pontos.
- **aspirar()** - Tenta aspirar a sala atual. Pode falhar **p_succao** vezes. Perde **v_aspirar** pontos. Se conseguir, ganha bonus **v_limpou**.
- **assoprar()** - Tenta assoprar a sujeira na sala local. Pode falhar **p_succao** vezes. Perde **v_assoprar** pontos. Se assoprar na sala de descarga com sucesso, ganha **v_desc** bonus.
- **passar_vez()** - Passa a vez. Perde **v_passarvez** pontos. Não falha. Não dá bônus.
  Todas acoes podem causar terremoto, com probabilidade dada **p_terremoto**, seja modelo básico ou total.

Sensores do Aspipó (retorna o valor do sensor):

- **ler_sujeira()** - Perde **v_ler** pontos. Retorna 1 se sujo, 0 se sala limpa. Pode falhar com probabilidade **p_sensorial**.
- **ler_posicao()** - Perde **v_ler** pontos. Retorna o numero da sala atual. Pode falhar com probabilidade **p_sensorial**.
- **ler_descarga()** - Perde **v_ler** pontos. Retorna 1 se esta for sala de descarga, e retorna 0 c.c. Pode falhar **p_sensorial** vezes.
- **ler_chamado()** - Perde **v_ler** pontos. Retorna o número de uma sala suja, ou -1 caso todas estejam limpas. Pode falhar **p_sensorial** vezes.
- **obs.terremoto** - Não é uma função, mas uma variável compartilhada da estrutura **observavel obs**. Sente a ocorrência de terremotos se igual a 1, e 0 cc. É de valor volátil, só permanecendo com seu valor durante uma ação. Não falha na leitura. Não é habilitada no nível de observabilidade máximo **OBS4**.
  Sensores podem causar terremotos se o nível de determinismo incluir **DETTERREMOTOTOTAL**.

Reações do Ambiente:

- Barra o Aspipó, se ele trombar na parede ao tentar andar para fora do limite.
- Limpa o ambiente, se o Aspipó aspirar. Exceto se aspirar a sala de descarga, que continua sempre suja. Pode falhar com a probabilidade configurada.
- Suja o ambiente, se o Aspipó assoprar. O Aspipo fica com o saco vazio. Pode falhar com a probabilidade configurada.
- Diz a condição de sujeira, se o Aspipó ler_sujeira(). Pode falhar com a probabilidade configurada.

- Diz a posição do Aspipó, se ele ler_posicao(). Responde o numero da sala. Pode falhar com a probabilidade configurada.
- Diz se a sala é de descarga, se ele ler_descarga(). Pode falhar com a probabilidade configurada.
- Diz o número de alguma sala suja, se o Aspipó ler_chamado(). Pode falhar com a probabilidade configurada.

Ações independentes do Ambiente:

- Aleatoriamente suja uma ou mais salas.
- Terremoto! Muda o Aspipó de sala e suja todas salas. O Aspipó sente na variável externa **obs.terremoto**.
- Calcula os pontos da medida de desempenho com pontos(). Pode mostrar os pontos durante a execução da simulação, se habilitado pelo nível de observabilidade OBS0, com a função **mostrar_pontos()**.
- Caso a opção **FOLGA1** seja habilitada, o ambiente estabiliza e não se suja mais a partir de MAXACOES do agente. Ficam desabilitados os terremotos. O agente passa a ter até a iteração MAXACOESFINAL para **finalizar_ambiente()**. Caso não tenha folga, com **FOLGA0**, o ambiente ignora as ações do agente entre MAXACOES e MAXACOESFINAL, tempo que o agente tem para **finalizar_ambiente()**. Caso o agente falhe de finalizar no tempo certo, sofrerá penalidades e o ambiente será abortado por segurança, para evitar laços infinitos.
- O agente pode continuar no ambiente pelo tempo que quiser, dentro dos limites, para limpar as salas restantes. O agente pode finalizar a qualquer tempo, mesmo antes de completar MAXACOES.

Objetivos:

- Objetivo abstrato: manter o ambiente limpo das impurezas.
- Objetivo formal: maximizar o valor da equação de medida de desempenho.
- Super objetivo: encontrar uma medida de desempenho que represente bem o objetivo abstrato para uma determinada configuração de mundo.

Índice