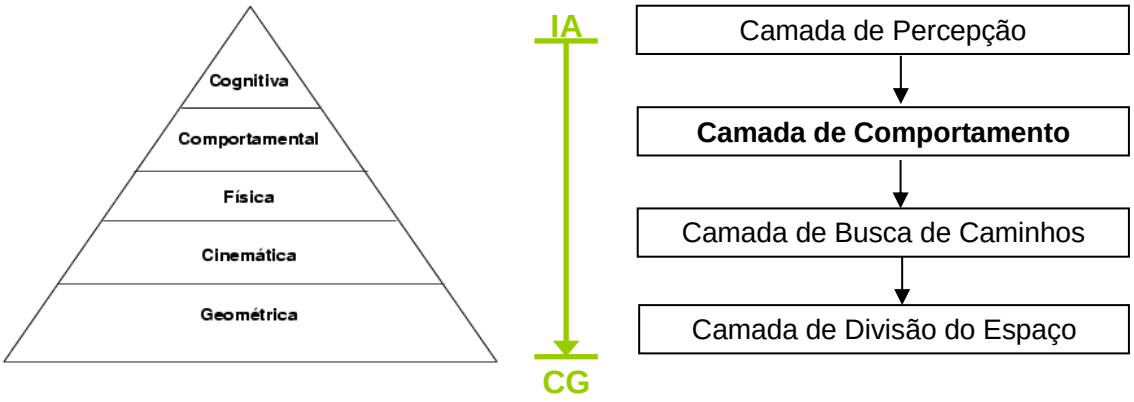
Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica Avançada
pozzer@inf.ufsm.br
10/10/2013

Seleção de Comportamentos



1. Maquina de Estados Finitos

Máquinas de Estados Finitos (*Finite State Machines* - FSM) são provavelmente o padrão de software mais utilizado em jogos para selecionar o **comportamento** de agentes reativos. Isso se deve a vários motivos [4]:

1. **Elas são rápidas e simples de implementar**: existem várias formas de implementar e todas são muito simples, como será visto ao longo deste material;
2. **São fáceis de depurar**: Isso ocorre principalmente quando possuir poucos estados;
3. **Gastam pouco processamento**: uma característica essencial da implementação de agentes reativos, cuja característica marcante é a resposta rápida a estímulos externos.
4. **São intuitivas**: Representam a forma como muitas vezes um humano raciocina. Também tem o papel de quebrar o comportamento de um personagem em blocos, onde cada bloco representa uma possível ação que pode ser desempenhada. Também facilita a discussão da implementação com o game designer, que muitas vezes não tem conhecimento de linguagens de programação.
5. **São flexíveis**: Pode ser facilmente ajustada pelo programador para prover comportamentos requeridos pelo *game designer*. Também permite que novos comportamentos sejam incluídos. Além disso, permite que sejam incorporados outros recursos como lógica *fuzzy* ou redes neuronais.

Apesar de possuir vários pontos a favor, também apresenta problemas. Isso ocorre quanto o número de estados torna-se muito grande e também porque elas são construídas de forma *ad hoc*, geralmente sem nenhum planejamento inicial de sua estrutura.

Historicamente, uma FSM era um dispositivo formalizado usado por matemáticos para resolver problemas. A FSM mais famosa é provavelmente a máquina de Turing. Ela foi a predecessora do atual computador e era capaz de realizar qualquer operação lógica por meio da leitura, escrita e remoção de símbolos em uma fita de tamanho infinito. Ela possuía uma cabeça de leitura/gravação que podia mover para a esquerda ou direita, dependendo do valor processado.

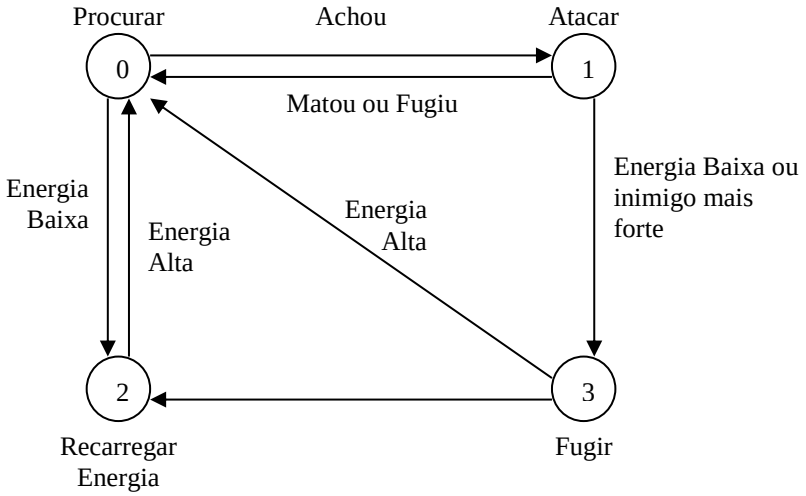
Uma máquina de estados é basicamente composta por um conjunto de estados e um conjunto de regras de transição entre estes estados (que usualmente refletem algum evento no mundo do jogo). A utilização desse modelo de representação e controle do comportamento de agentes consiste basicamente em representar, através dos estados, as ações possíveis ao agente, de forma que as regras de transição representem as condições que ao

serem verificadas para avaliar se o agente deve mudar de estado [1]. A idéia é decompor o comportamento de um objeto em ações ou estados que possam ser facilmente tratados.

Um exemplo bem simples de uma FSM é um interruptor de luz. A lâmpada pode estar ligada ou desligada. Se estiver ligada, pode permanecer neste estado ou ser desligada. Se estiver desligada, ela pode se ligada. Em jogos, uma FSM não é tão simples assim, visto que geralmente possuem mais de dois estados. A seguir são apresentados alguns exemplos de FSM em jogos [4]:

- Os fantasmas *Inky*, *Pink*, *Blink* e *Clyde* do jogo Pac-man são implementados via FSM. Os fantasmas possuem 2 comportamentos principais: caçar (*Chase*) e fugir (*Evade*) do adversário. A implementação da ação caçar de cada fantasma é diferente. A transição de estados ocorre sempre que o jogador conseguir alguma pílula de energia;
- Os bots do jogo Quake também usam FSM. Eles possuem estados como procurar armadura (*FindArmor*), procurar por kit médico (*FindHelth*) e correr (*RunAway*). Até mesmo as armas possuem uma mini FSM, como no caso de um míssel que implementa estados como mover (*Move*), tocar objeto (T*ouchOject*) e morrer (*Die*);
- Os jogadores do FIFA2002 também fazem uso de FSM. Alguns estados são: driblar (*Dribble*), correr atrás da bola (*ChaseBall*) e marcar jogador (*MarkPlayer*). Além disso, os times também implementam estados, como defender (*Defend*) ou sair do campo (*WalkOutOnField*);
- Os NPCs do jogo Warcraft usam estados como mover para posição (*MoveToPosition*), patrulhar (*Patrol*) e seguir um caminho (*FollowPath*). A implementação destes diversos estados citados será vista módulo do curso que trata de comportamentos.

A seguinte figura mostra um exemplo de uma máquina de estados (4 estados e 7 funções de transição) representando o comportamento de um agente inteligente simples em um jogo do gênero *First Person Shooter* (FPS).



Observe que neste diagrama não estão especificados como será implementada a função de busca do inimigo, nem como deve ser o ataque. Refinamentos sucessivos, bem como o uso de outras técnicas podem ser utilizados a este propósito.

Na execução do comportamento, a máquina de estados encontra-se inicialmente em seu estado inicial; a cada iteração, as regras das transições que deixam o estado corrente são avaliadas; se alguma delas for disparada, a transição é então realizada, e o estado de chegada desta regra se torna o novo estado corrente. As ações associadas ao estado serão então executadas pelo agente [1]. A máquina pode estar em único estado a cada momento.

```
void run( int *state )
{
   switch( *state )
   {
      case 0:  //procurar inimigo
         procurar();
         if( encontrou_inimigo )
         *state = 1;
         break;
      case 2:  //recarregar energia
         recarregar();
         if( energia > 90 )
         *state = 0;
         break;
```

```
case 1:  //atacar inimigo
   atacar();
   if ( morto )
   {
      morrer();
      *state = -1;
   }
   if ( matou )
   {
      *state = 0;
   }
   if( energia < 50 || inimigo_ou_arma_forte)
      *state = 3;
   break;
```

```
case 3:  //fugir
   fugir();
   if( não encontrou_inimigo )
   {
      if( energia < 50 )
         *state = 2;
      else
         *state = 0;
   break;
   }
}
```

Esta estartégia de implementação é a mais simples, porém é que apresenta maiores problemas quanto o número de estados torna-se muito grande. O uso de Linguagens de especificação de FSM [Rabin – Wisdom, pg 314] por meio de macros (#defines), é apontado como uma solução mais adequada.

2. Maquina de Estados Finita Hierárquica

Á medida que a complexidade do comportamento dos agentes aumenta, as FSMs tendem a crescer de forma descontrolada, o que torna a implementação impraticável. Eis alguns problemas do uso de FSM:

- É uma ferramenta poderosa, mas pode se tornar terrivelmente complexa quanto leva em consideração ações muito básicas necessárias a um agente;
- A representação visual torna-se intratável;
- Comportamentos complexos necessários em jogos modernos
- Um grande número de estados pode tornar a FSM ilegível (Veja Figura 2);

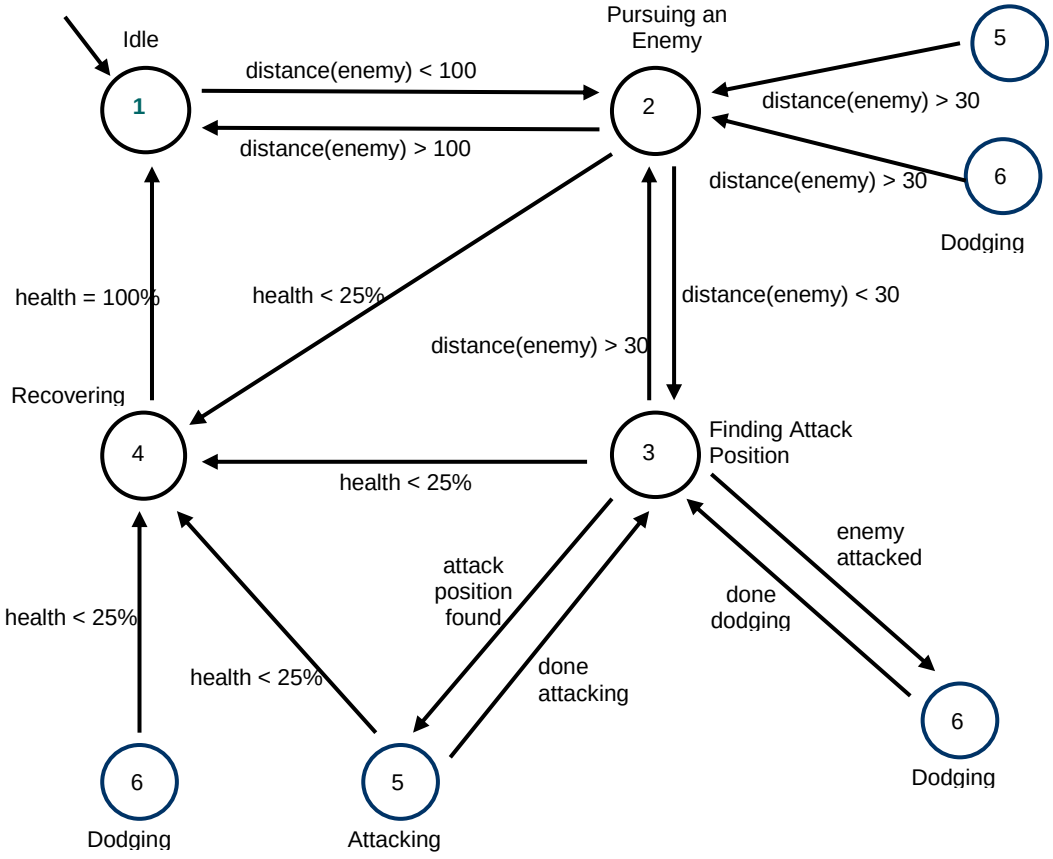


Figura 2: Exemplo de uma FSM com muitos estados [5]

Uma solução possível para este problema é o uso de máquinas hierárquicas (HFSM), também chamadas *Behavioral Transition Networks* [Houlette_01], como proposto por vários pesquisadores [Santos_04, Fu_04, Orkin_02]. Nesta abordagem, níveis mais altos lidam com ações mais genéricas, enquanto níveis mais baixos lidam com ações mais específicas (cada estado pode ser uma nova FSM). Entretanto, a hierarquia nem adiciona mais poder ao modelo, nem reduz o número de estados. Ela pode somente reduzir significativamente o número de transições e tornar a FSM mais intuitiva e simples de compreender. É importante observar que qualquer HFSM pode ser reescrita como uma FSM sem hierarquia [Harel_87].

3. Maquina de Estados Finita com Pilha (Stack-based FSM)

Geralmente as ações a serem realizadas possuem pré-condições. Essas pré-condições são expressas por meio de ações auxiliares que devem ser realizadas para garantir a integridade e veracidade da ação previamente estabelecida (ação terminal). Quando se está trabalhando com a representação de ações complexas e abstratas, onde a execução de uma ação depende da execução de um conjunto de ações precedentes, HFSMs apresentam as mesmas limitações que as FSMs. Como exemplo, suponha que um agente tem como objetivo a libertação de outro personagem que está raptado. Para que isso ocorra pode ser necessário, por exemplo, que o personagem deva executar um conjunto de ações auxiliares, como andar, lutar, conversar, dentre outras. Muitas destas ações podem envolver outros agentes que habitam o mesmo ambiente. Quando o personagem cumprir todos os pré-requisitos necessários, o agente pode então realizar a ação à qual está designado.

Para lidar com estes tipos de situações, pode-se acoplar uma estrutura de pilha a cada agente como memória auxiliar à FSM, de modo a evitar a criação de nós adicionais que seriam necessários para tratar situações onde ações possuem como pré-condições a execução de outras ações. A pilha possui apenas duas posições. Na base (*slot 0*) é guardada a ação que representa o objetivo final (*goal*) do agente, ou seja, o evento que representa o fim da missão, que, para o exemplo anterior, é a libertação do prisioneiro. No topo da pilha (*slot 1*) é armazenada a ação intermediária corrente. A meta do agente é sempre executar a ação que está no topo da pilha, que é sempre a última inserida. Uma ação permanece na pilha enquanto sua execução estiver ocorrendo ou até que outra ação mais importante (geralmente pré-condição) faça-se necessária.

Na Figura 4 é apresentada a arquitetura (Criada para atender aos requisitos de uma aplicação de *storytelling* [2], [3]), composta basicamente por 5 módulos, usada no controle dos comportamentos dos agentes:

- *Perceptor*: Continuamente adquire informação do mundo. O sistema visual do agente realiza testes de colisão com objetos na cena. O sistema de mensagens processa mensagens enviadas por outros agentes. Estas mensagens podem mudar o estado interno de modo que o agente possa participar de uma ação coletiva;
- Pilha de duas posições: A pilha é usada como um repositório de ações que devem ser executadas;
- Fila de história: Mantém um histórico das últimas N ações executadas. Com este repositório, no caso da inexistência de ações específicas, o Gerenciador de Ações pode selecionar ações aleatórias não repetitivas para o agente. A pilha, juntamente com a fila, representam a memória de trabalho do agente;
- Gerenciador de Comportamentos (FSM): o elemento central da arquitetura. Ele processa entradas, determina novas ações auxiliares e designa ações ao *effector* que melhor refletem o estado corrente do agente;
- *Effector*: É o meio pelo qual o agente muda o ambiente. Mensagens podem invocar outros agentes e, conseqüentemente, mudar seus atributos. Além disso, este módulo é o responsável pela movimentação do agente.

A memória de trabalho do agente é representada pela pilha e fila de história. O estado corrente do agente é representado unicamente pela pilha. Tanto o Gerenciador de Ações, como o próprio agente, podem inserir operações na pilha. O Gerenciador sempre insere uma nova ação terminal na base toda vez que a pilha estiver vazia. O próprio agente deve comunicar ao gerenciador que não tem mais ações a executar. Garantindo que as ações terminais são sempre inseridas na base, a pilha ainda tem uma posição livre para tratar ações auxiliares que se fizerem necessárias. Essas ações auxiliares são sempre inseridas pelo próprio agente, mediante a impossibilidade do cumprimento da ação terminal, segundo regras implementadas em cada nó da FSM. Como exemplo, se o agente estiver se locomovendo, ações candidatas correspondem a outras ações de deslocamento, no caso, para evitar colisão com possíveis objetos. Se o agente deve atacar um determinado local, ações de deslocamento e luta podem ser continuamente inseridas até que a ação de ataque seja finalizada.

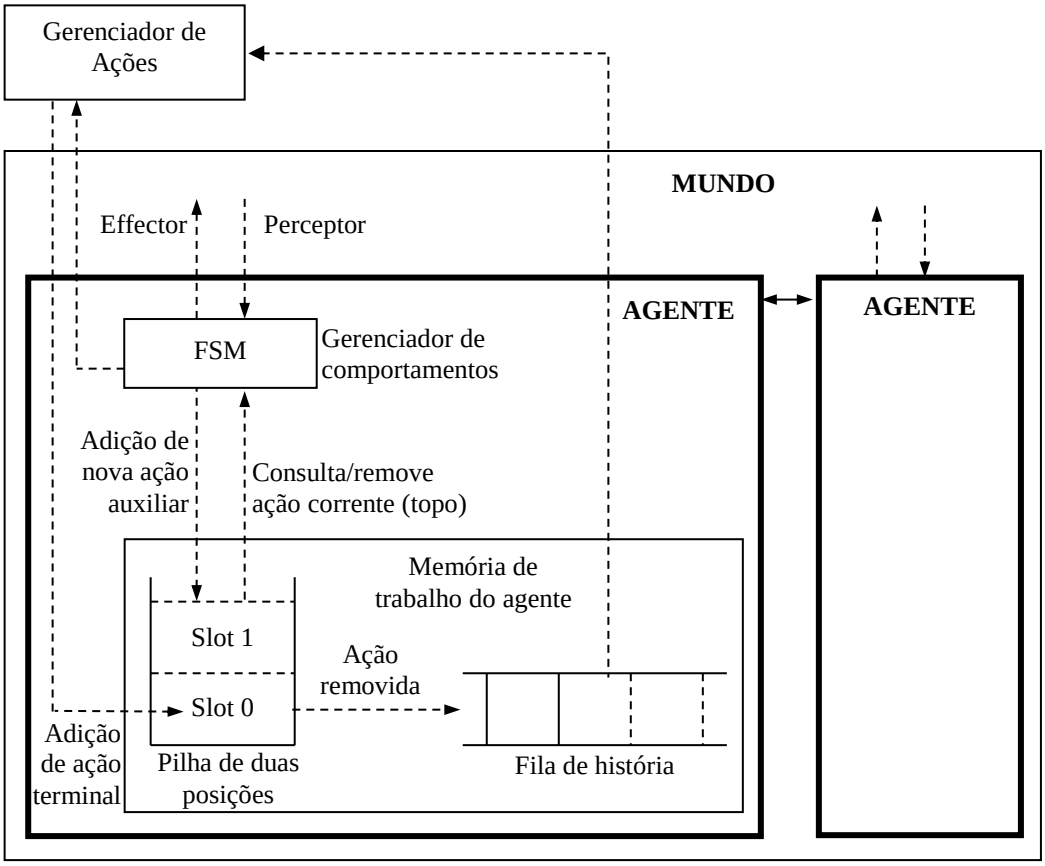


Figura 4: Exemplo de uma FSM com Pilha [2]

Na Figura 5 é apresentado um comportamento simplificado da pilha para o exemplo da libertação da vítima. No estado inicial, o herói recebe como ação terminal a libertação da vítima. Esta ação é então armazenada na base da pilha do herói, e somente será removida com o final de sua execução. No estado *Free* da FSM, duas pré-condições estão associadas: o herói deve estar próximo da vítima e deve derrotar o vilão (raptor da vítima). Quando a FSM do herói é executada, as pré-condições são avaliadas. Supondo que ele já esteja no mesmo local da vítima, este cria uma ação auxiliar para lutar com o vilão, que é colocada no topo da pilha. A ação de lutar, por sua vez, tem como pré-condições estar próximo e ser mais forte que o oponente. Caso os personagens estejam próximos e o vilão seja mais forte que o herói, uma ação para tornar o herói mais forte é adicionada em sua pilha pelo estado *Fight* da FSM. Neste momento, a pilha já está com as duas posições ocupadas (pilha cheia). Neste caso, a ação do topo *Fight* é removida para a inserção da nova (*Get_stronger*), mesmo que esta ainda não tenha sido finalizada. Quando a operação *Get_stronger* finalizar, a execução volta à base da pilha (*Free*), o que faz com que a operação *Fight*, por ainda não ter sido executada e por ser pré-condição da libertação, seja novamente reinserida no topo da pilha, desta vez com suas pré-condições já satisfeitas. A ação *Fight* é então executada até que o herói derrote o vilão. Quando o *Fight* é finalizado, a ação *Free* pode então ser executada. Quando a operação *Free* é finalizada, o personagem não tem nenhuma ação a realizar. Neste momento, ele solicita ao Gerenciador o envio de uma nova ação.

Em termos gerais, o agente deve ter a habilidade de parar a execução da ação corrente em favor da execução de uma ação de mais alta prioridade. Isso não trás nenhum problema, porque, se for necessário, a ação removida pode ser reinserida na pilha pela FSM tantas vezes quanto forem necessárias.

Fazendo-se uma analogia à máquinas hierárquicas, cada posição da pilha age como uma FSM, que armazena a configuração local do contexto sendo executado. A pilha não necessita ter mais do que duas posições, pois com um *slot* variável é possível armazenar uma ação auxiliar. Ações auxiliares, resultantes de outras ações auxiliares não necessitam de espaço dedicado, visto que a partir da ação da base é possível reconstruir a hierarquia de eventos necessários a sua execução. Devido a esta estratégia, pode haver um pequeno overhead na reinserção de ações auxiliares que foram removidas em função de outras ações auxiliares mais atuais.

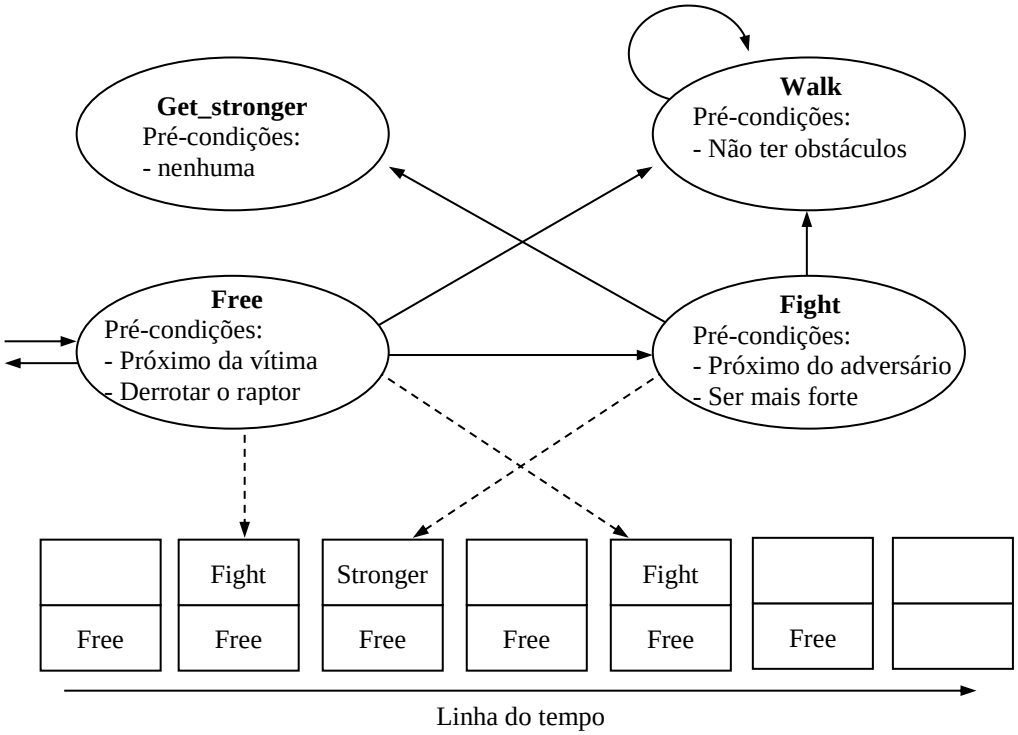


Figura 5: Exemplo do gerenciamento de ações complexas

O uso de mais de duas posições pode levar a situações onde ações auxiliares, mesmo não mais válidas e necessárias, sejam executadas. Como exemplo, suponha que em um dado momento, o agente está executando a ação A1 que consiste em ir a um determinado local para desviar de um obstáculo. Neste mesmo tempo, ele detecta a presença de um inimigo que o faz fugir para uma direção perpendicular. Após a fuga, uma nova rota ao destino final deve ser criada. Se A1 ainda estivesse na pilha, estaria no topo da pilha e deveria ser executada. Entretanto, após a fuga, a ação A1 pode estar totalmente obsoleta, o que poderia resultar em uma ação sem lógica do ponto de vista do usuário. Neste presente trabalho, todos os exemplos testados indicaram que uma pilha com apenas duas posições é a melhor solução. De fato, na arquitetura proposta, ações complexas como "perseguir um inimigo, desviando de obstáculos, para iniciar um combate", podem ser adequadamente tratadas.

Este mecanismo de pilha permite também que após o término da ação terminal, novas ações terminais sejam inseridas pelo próprio agente. Isso pode ocorrer em várias circunstâncias. Pode-se especificar que, após uma libertação, por exemplo, ambos os personagens devem ir para suas respectivas moradias.

Uma abordagem que também faz uso de uma pilha como forma de expandir as potencialidades das FSM foi proposta por [Tozour – Wisdom2, pg 303], com a diferença que a pilha tem tamanho ilimitado. Como exemplo, consideremos a FSM da Figura 6. Neste caso, na base da pilha existe uma ação *idle*, que é executada quando o personagem (guarda) não está realizando nada. O guarda pode então decidir fazer um patrulha e então procurar pelo inimigo. Caso encontrar o inimigo, ele pode começar a lutar e após derrotar inimigo, voltar ao estado de procura. Observe que não é necessário se ter uma transição do estado *Combat* para outros, visto que quanto o combate terminar, a ação logo abaixo na pilha será executada automaticamente até que seja finalizada.

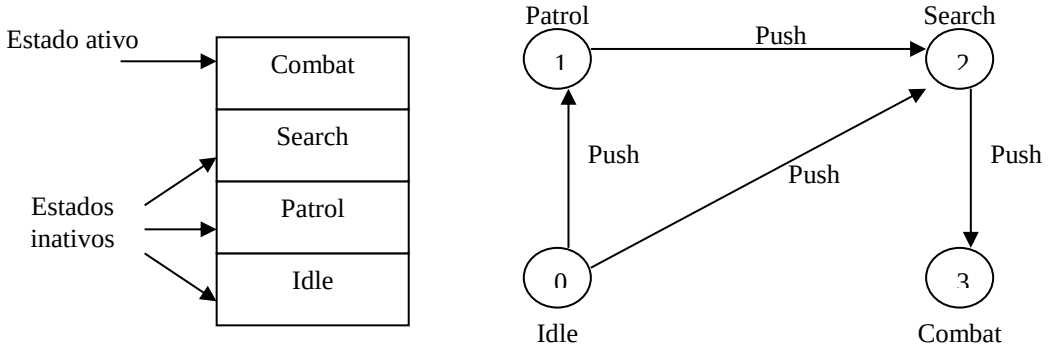


Figura 6: Exemplo de uma pilha para uma FSM [Tozour – Wisdom2, pg 303]

No Algoritmo 2 é ilustrado um pseudo-algoritmo para implementar o esquema de pilha apresentada na Figura 4.

```
void Actor::run()
{
   currAction = getCurrentAction();

   switch( currAction->actionType )
   {
      case ACTION_STAND:
         finishedAction = stand();
         break;

      case ACTION_WALK:
         finishedAction = walk();
         break;

      case ACTION_ATTACK:
         finishedAction = attack();
         break;

      case ACTION_FIGHT:
         finishedAction = fight();
         break;
   }

   if( finishedAction == true )
   {
      endCurrentAction();
   }
}
```

Algoritmo 2: Exemplo de algoritmo de FSM com pilha

No Algoritmo 3 são apresentadas a implementação simplificada de alguns estados da máquina para este exemplo.

```
bool Actor::fight()
{
   float dist = actorPos.distance(enemy->actorPos );

   if( dist > 100 )
   {
      setNewAction( 1, new Action( ACTION_WALK, enemy->actorPos, t1 ));
      return walk();
   }

   //codigo da luta

   if( matou_inimigo )
   {
      setNewAction( 1, new Action( ACTION_WALK, home, t1 ) );
      setNewAction( 0, new Action( ACTION_STAND, t1, t2 ) );
      //return true;
   }
   return false;
}

bool Actor::walk()
{
   actorPos = actorPos + actorDir*actorSpeed;

   if( dist < 100  )
   {
      return true;
   }
   return false;
}
```

Algoritmo 3: Implementação dos estados Fight e Walk

4. Adicionando Mensagens a FSM

Jogos bem projetados tendem a ser baseados em eventos (*event driven*). Isso significa que quando um evento ocorre, ele é transmitido (*broadcast*) para os objetos relevantes do jogo de modo que eles possam responder de modo adequado. Esses eventos geralmente são transmitidos em formado de um pacote que contem informações sobre o evento tais como quem enviou, quem deve responder a ele, a hora que o evento ocorreu, dentre outros [4].

Uma implementação *event-driven* trás vantagens como o ganho de desempenho. Sem um tratador de eventos (*event handling*), os objetos devem ficar continuamente analisando o jogo para saber se uma situação particular ocorreu. Com um sistema de *event handling*, o objeto é notificado toda vez que uma mensagem for direcionada a ele. Se a mensagem for pertinente, ações podem ser tomadas.

Agentes de jogos também podem se utilizar desta tecnologia para se comunicarem entre si. Com recursos de envio, tratamento e resposta de mensagens, pode-se implementar vários comportamentos como:

- Um fuzileiro disparou um tiro contra um soldado: Uma mensagem é enviada para o soldado indicando que ele levou um tiro com intensidade I;
- Um jogador de futebol cruzou a bola para um companheiro: O jogador que fez o passe pode dizer ao companheiro em que direção deve correr para pegar a bola e em que momento;
- Pedido de socorro: um agente que esteja sofrendo um ataque pode solicitar ajuda a um médico. Quando o primeiro médico chegar, uma nova mensagem pode ser enviada dizendo que o socorro chegou e os demais que estavam indo ajudá-lo podem voltar a sua atividade anterior.

A mensagem pode ter vários formatos. Pode ser uma simples constante ou pode ser uma estrutura com vários parâmetros. Nos parâmetros pode-se incluir a hora que a mensagem foi passada, quem enviou, qual o destinatário, a mensagem, dentre outros. A mensagem pode incluir uma localização, uma força, um efeito, dentre outros, como mostrado no Algoritmo 4.

```
enum MSG_Name {MSG_Attacked=1, MSG_Damaged, MSG_Poisoned };

class MSG_object
{
   public:
     MSG_Name name;
     float data;
     objectID sender;
     objectID receiver;
     float deliveryTime;
     ???   aditionalInfo;

   //métodos
      ....

}
```

Algoritmo 4: Definição de um objeto que representa uma mensagem

As mensagens podem ser passadas diretamente entre os agentes (caso todos os agentes tenham uma referência para os outros) ou para um gerenciador, que tem o papel de processar (distribuir) a mensagem, baseado nas informações presentes no objeto mensagem.

Deve-se tomar o cuidado para sincronizar o processamento das mensagens, observando-se que cada agente executa seqüencialmente no loop principal do jogo. Pode ocorrer de dois agentes atirarem um contra o outro simultaneamente (dentro do mesmo loop) e o agente que atirou primeiro mate o outro antes mesmo que o outro passe a mensagem indicando que também deu um tiro.

A estrutura de implementação de uma FSM torna o processo de comunicação entre agentes uma tarefa fácil. Dentro da implementação de cada comportamento, pode-se enviar mensagens pertinentes a outros agentes.

```
bool Actor::fight()
{
```

```
   Actor *enemy = GetActor(enemyID);
   float dist = actorPos.distance(enemy->actorPos );

   if( dist > 100 )
   {
      setNewAction( 1, new Action( ACTION_WALK, enemy->actorPos, t1 ));
      return walk();
   }

   MSG_object *msg = new MSG_object(this, enemy, MSG_Attacked, 20);
   bool = sendMessage(msg);

   if( bool == 0 ) //matou inimigo
   {
      setNewAction( 1, new Action( ACTION_WALK, home, t1 ) );
      setNewAction( 0, new Action( ACTION_STAND, t1, t2 ) );
      //return true;
   }
   return false;
}

bool Actor::onMessage(MSG_object *msg)
{
   if( ! alive() )
   {
      return MSG_NO;
   }
   //processa a mensagem
   // - pode deduzir a energia
   // - pode ser um chamado
   // - pode aumentar a energia
   // - etc.
}
```

Observações:

- O método `GetActor()` pode pertencer a uma classe que gerencia todos os personagens do jogo (classe `EntityManager`). Essa classe pode aceitar a inclusão de novos personagens, remoção e tem como principal função retornar uma referência para o personagem que possui o ID informado;
- O método `sendMessage()` pode pertencer a uma classe que gerencia o envio de mensagens entre os personagens (classe `MessageDispatcher`). As mensagens podem ser enviadas imediatamente ou com um certo *delay*, dependendo das informações contidas no objeto `Message`. Deve-se observar que todas as mensagens e ações devem estar vinculados a *time stamps* (temporizadores);
- O método `onMessage()` está na implementação da classe `Actor`, e em uma estrutura de classes bem projetada, como método virtual da classe pai de todos os personagens e/ou entidades do jogo. Recomenda-se o uso de diagramas UML para a modelagem da hierarquia e relacionamento de classes de um jogo.

5. Sistema Baseado em Regras

Uma outra forma de representação de comportamento muito utilizada em jogos, são os chamados sistemas baseados em regras (*Rule Based Systems* - RBSs). RBSs apresentam algumas vantagens como [1]:

- correspondem ao modo como as pessoas normalmente pensam sobre conhecimento;
- são bastante expressivos e permitem a modelagem de comportamentos complexos;
- modelam o conhecimento de uma maneira modular (como por exemplo organizando a base por objetivos);
- são fáceis de escrever (e depurar, quando comparados a árvores de decisão por exemplo); e
- são muito mais concisos que máquinas de estados finitos.

Em um RBS, o conhecimento é definido através de um conjunto de parâmetros (variáveis) e um conjunto de regras que trabalham sobre esses parâmetros, de modo que durante a tomada de decisão essas regras são então processadas [1]. O conhecimento está expresso por meio de um conjunto de regras *if-then*, onde a condição representa o conjunto de cláusulas condicionais ligadas por conjunções (conectivos **E** e/ou **OU**) que denotam as

condições do jogo para que a regra seja aplicada, e a conclusão representa a ação a ser tomada pelo agente, dado o estado corrente do jogo.

Como exemplo, se o conjunto de parâmetros são distância e energia, e o conjunto de ações possíveis são procurar, recuperar e atirar, as seguintes regras de inferência podem ser utilizadas para determinar o comportamento de um agente:

```
if( inimigo está distante and saúde é alta )
   then procurar inimigo
if( saúde é baixa )
   then recuperar_energia
if( inimigo está próximo and saúde é alta )
   then atirar
```

Durante a definição do comportamento, as regras são avaliadas na ordem que foram definidas. Se todas as cláusulas de uma regra forem verdadeiras, então a execução do personagem corresponde à conclusão desta regra, senão a próxima regra é avaliada do mesmo modo. Dependendo do número de regras, este processamento pode se tornar muito caro. Além disso, pode ocorrer de alguma regra nunca ser verdadeira, caso os parâmetros não sejam corretamente estipulados.

6. Aprendizado por Árvores de Decisão

Árvores de decisão (*Decision Trees*) são ferramentas que podem ser utilizadas para dar ao agente a **capacidade de aprender**, bem como para **tomar decisões**. A idéia de aprendizado é que os perceptors (elementos do agente que percebem o mundo) não sejam usado apenas para agir, mas também para aumentar a capacidade do agente de agir no futuro [9]. O aprendizado ocorre na medida que o agente observa suas interações com o mundo e seu processo interno de tomada de decisões. Aprendizado de árvores de decisão é um exemplo de aprendizado indutivo: Cria uma hipótese baseada em instâncias particulares que gera conclusões gerais.

Árvores de decisão são similares a regras if-then. É uma estrutura muito usada na implementação de sistemas especialistas e em problemas de classificação. Tomam a decisão baseadas em um **conjunto de parâmetros de entrada, aplicado sobre um conjunto de regras**, dispostos nos nós de um árvore. Os atributos de entrada podem ser discretos ou contínuos. A busca inicia na raiz da árvore e para cada nó, seleciona-se o nó filho baseado no valor de um parâmetro de entrada. A busca é finalizada quando encontra-se um nó que atenda aos parâmetros de entrada. Para os exemplos tratados, serão considerados apenas valores discretos. O aprendizado de valores discretos é chamado classificação [9].

A árvore de decisão chega a sua decisão pela execução de uma seqüência de testes. Cada nó interno da árvore corresponde a um teste do valor de uma das propriedades, e os ramos deste nó são identificados com os possíveis valores do teste. Cada nó folha da árvore especifica o valor de retorno se a folha for atingida.

Como regra, o ideal é ter próxima a raiz as regras mais importantes, ou seja, as que melhor classificam a entrada. Com isso, espera-se resolver o problema aplicando-se o menor número de regras.

Para melhor compreender o funcionamento de uma árvore de decisão, vamos considerar o exemplo da Figura 7. Considera-se o problema de esperar para jantar em um restaurante. O objetivo é aprender a definição para *DeveEsperar*. Para qualquer problema de árvore de decisão, deve-se inicialmente definir atributos disponíveis para descrever exemplos de possíveis casos do domínio. São adotados os seguintes atributos: alternar de restaurante, ir para um bar, dia sa semana, estar com fome, número de fregueses, preço da comida, clima, se foi feita reserva, tipo do restaurante, estimativa de espera. Os atributos preço e tipo de comida, por serem de pouca importância, foram desconsiderados.

Em uma árvore de decisão, o conhecimento é representado em cada nó que, ao ser testado, pode conduzir a busca a um de seus filhos. Deste modo, descendo da raiz em direção as folhas da árvore, pode-se selecionar a configuração do sistema, e deste modo o comportamento associado [8].

Como segundo exemplo, pode-se atribuir ao nó raiz uma classificação do tipo de personagem, que pode pertencer ao mesmo time ou ser um adversário. O segundo nível pode se classificar de acordo com o tipo de

arma sendo utilizada, e as folhas como o comportamento associado a cada configuração. Por exemplo, se for encontrado um inimigo com uma arma potente, a ação indiciada é fugir (*Flee*).

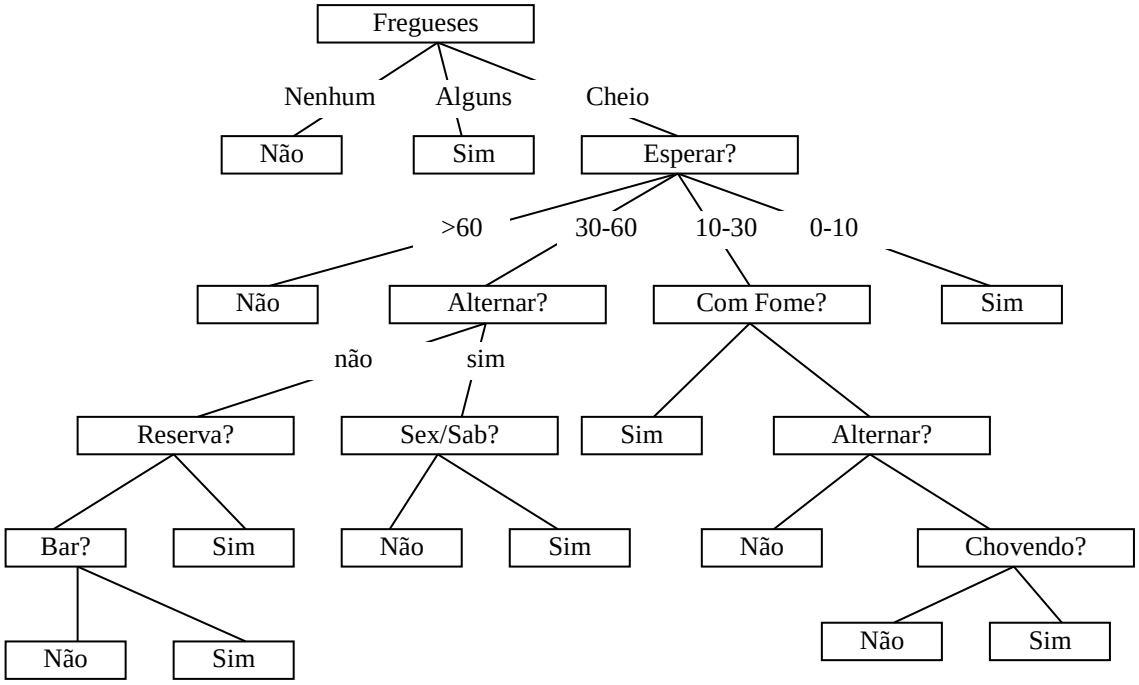


Figura 7: Exemplo de uma árvore de decisão para o problema de espera para jantar em um restaurante [4].

Como terceiro exemplo, vamos considerar a situação de ir ao trabalho pela manhã. Deseja-se saber qual o tempo estimado da viagem. A Figura 8 apresenta a árvore de decisão para este problema. Mas surge uma pergunta: como esta árvore e a árvore da Figura 7 são geradas? A resposta é a partir de exemplos de caso do domínio. A Tabela 1 ilustra alguns exemplos de casos de domínio para o problema da Figura 8.

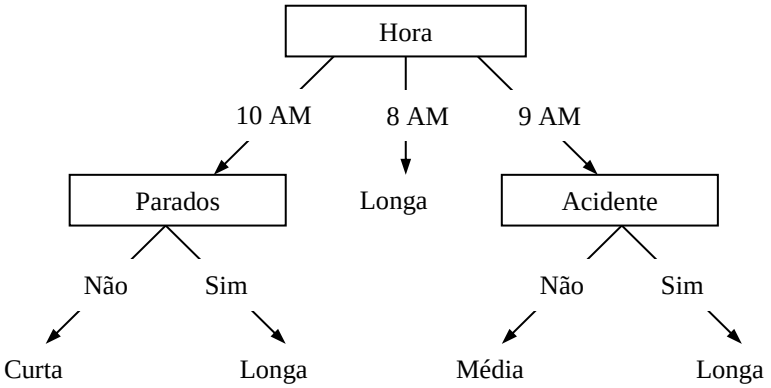


Figura 8: Árvore de decisão para duração de viagem esperada

Tabela 1: Exemplos de experiências de direção no trânsito

Amostra	Atributos				Alvo
	Hora (AM)	**Clima**	**Acidentes**	**Parados**	**Duração**
D1	8	Sol	Não	Não	Longa
D2	8	Nuvem	Não	Sim	Longa
D3	10	Sol	Não	Não	Curta
D4	9	Chuva	Sim	Não	Longa
D5	9	Sol	Sim	Sim	Longa
D6	10	Sol	Não	Não	Curto
D7	10	Nuvem	Não	Não	Curto
D8	9	Chuva	Não	Não	Médio
D9	9	Sol	Sim	Não	Longa

D10	10	Nuvem	Sim	Sim	Longa
D11	10	Chuva	Não	Não	Curta
D13	8	Nuvem	Sim	Não	Longa
D13	9	Sol	Não	Não	Médio

A grande questão desta tecnologia é como a árvore pode ser gerada, ou seja, como escolher as regras mais importantes e quais regras podem ser descartadas da árvore. Como regra, o ideal é que a árvore tenha as regras mais importantes próximas a, ou seja, as que melhor classificam a entrada. Com isso, espera-se resolver o problema aplicando-se o menor número de regras.

```
Node criaArvore (exemplos, atributoAlvo, atributos)
{
    se todos exemplos tem o mesmo valor de atributoAlvo
       retorna a folha com o valor
    senão se o conjunto de atributos é vazio
       retorna a folha com o valor de atributoAlvo mais comum entre os exemplos
    senão
    {
      BEST = melhor atributo entre atributos com um variações v1, v2, .. vk
      Particione os exemplos segundo seus valores para A em conjuntos S1, S2, ...Sk
      Crie um nó de decisão N com atributo BEST
      Para i=1 até K
          Conecte um no B para o nó N com teste vi
          Se Si tem elementos (não vazio)
             Conecte ramo B a criaArvore(Si, atributoAlvo, atributos – { BEST })
          Senao
             Conecte B para a folha do nó com atributoAlvo mais comum
      Retorna no N
   }
}
```

Algoritmo 5: Algoritmo de aprendizado para árvores de decisão

Quando a função é chamada, passam-se todos os elementos do conjunto de treinamento (D_1, D_2, ... D_n), o atributo alvo (Duração) e o conjunto de atributos disponíveis para serem escolhidos (Hora, Clima, Acidente, Parados). O algoritmo escolhe o ***melhor atributo*** para repartir as instâncias e criar o nó de decisão correspondente. A Figura 9 mostra a árvore de decisão depois da escolha do atributo hora como o melhor atributo para dividir os exemplos. Cada nó possui um conjunto com os exemplos restantes e um histograma dos exemplos de acordo com o atributo alvo. Por exemplo, para a raiz temos 4 curtos, 2 médios e 7 longos.

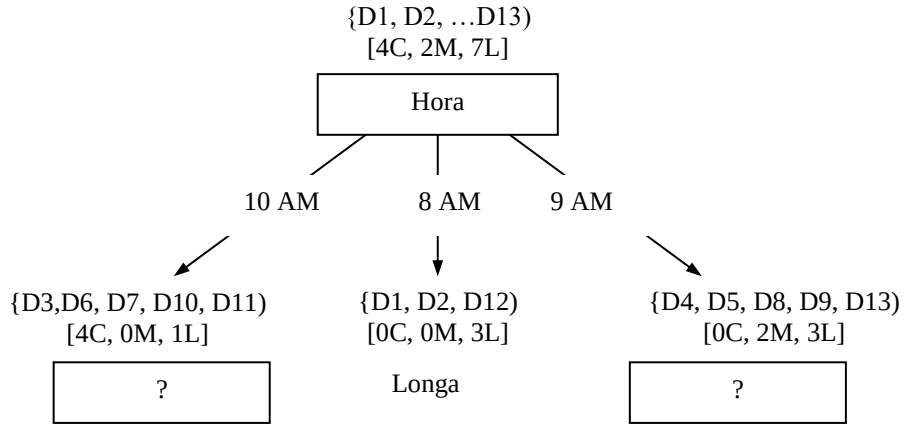


Figura 9: Árvore de decisão parcialmente construída, com o atributo hora na raiz.

A recursão do algoritmo pára quando uma das três condições for verdadeira:

- Todos os exemplos têm o mesmo atributo alvo:
- Não existem mais atributos
- Não existem mais exemplos.

O algoritmo heurístico mais conhecido para a escolha do melhor atributo é o ID3. Ele é baseado no cálculo da entropia, ou seja, na escolha inicial de atributos que minimizem a entropia, a qual quantifica variação em um conjunto de exemplos em relação aos valores do atributo alvo.

O primeiro jogo de entretenimento a utilizar árvores de decisão com sucesso foi o *Black&White*. Neste jogo as ADs são utilizadas para representar as informações sobre as experiências que a criatura tem ao longo do jogo, por exemplo, as experiências sobre que tipos de objetos que foram comidos. Desta forma, em ocasiões futuras, a criatura é capaz de tomar decisões sobre que tipo de objetos que são mais apropriados para comer [7]. A IA aprende por experiência, ou seja, analisando jogadas já realizadas pelo jogador, e a que direção cada jogada conduziu.

Toda vez que a criatura faz alguma coisa, ela grava as reações do jogador e usa a tupla (ação, resposta do jogador) como entrada no mecanismo de indução para construir a árvore de decisão que guia a escolha de ações futuras. Assim, a criatura tende a realizar ações que foram recompensadas pelo jogador, e evitar as demais.

6.1 Algoritmo ID3

Por Cícero de Lara Pahins, 2013

O algoritmo heurístico ID3 é baseado no cálculo da entropia, ou seja, na escolha inicial de atributos que minimizem a entropia, a qual quantifica variação em um conjunto de exemplos em relação aos valores do atributo alvo.
A entropia de um conjunto é definida como sendo o grau de pureza deste conjunto, assim temos que:

$$Entropia(S) = -p_+ log_2\, p_+ - p_- \, log_2 p_-$$

onde

S é conjunto de treinamento
p_+ é a porção de exemplos positivos (sim)
p_- é a porção de exemplos negativos (não)

Para melhor entender o significado desta fórmula podemos calcular a entropia para o exemplo de um saco com 6 bolas, pretas (exemplos negativos) e brancas (exemplos positivos), nos seguintes casos:

1. Todas as bolas são brancas (6)

$$\begin{aligned} Entropia(S) &= -\left(\frac{6}{6}\right) log_2 \left(\frac{6}{6}\right) - \left(\frac{0}{6}\right) log_2 \left(\frac{0}{6}\right) \\ &= -(1)0 - (0)0 = 0 \end{aligned}$$

2. 3 bolas pretas e 3 bolas brancas

$$\begin{aligned} Entropia(S) &= -\left(\frac{3}{6}\right) log_2 \left(\frac{3}{6}\right) - \left(\frac{3}{6}\right) log_2 \left(\frac{3}{6}\right) \\ &= -(0.5)(-1) - (0.5)(-1) = 1 \end{aligned}$$

Observamos que no caso extremo em que todas as 6 bolas são iguais a entropia tem valor 0, ou seja, mínimo, enquanto que possui valor máximo 1 no caso de uma distribuição 50-50%.

Agora que definimos o conceito de entropia, devemos buscar uma forma de calcular a sua redução conforme a escolha de diferentes atributos na árvore de decisão. Chamamos isto de Ganho, o qual é definido como a redução esperada na entropia de S, ordenado pelo atributo A, dado pela seguinte equação:

$$Ganho(S, A) = Entropia(S) - \sum_{v \in valores(A)} \frac{|S_v|}{|S|} Entropia(S_v)$$

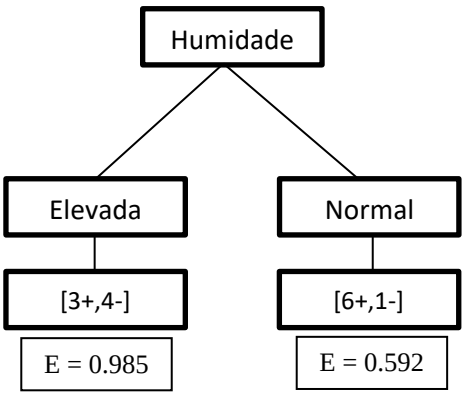
Deste modo, o algoritmo ID3 busca o atributo que apresente o maior ganho a cada iteração. Abaixo iremos construir um exemplo para a construção da árvore de decisão de jogar tênis (CFBioinfo, 2013):

Dia	Aspecto	Temp.	Humidade	Vento	Jogar Tênis
D1	Sol	Quente	Elevada	Fraco	Não
D2	Sol	Quente	Elevada	Forte	Não
D3	Nuvens	Quente	Elevada	Fraco	Sim
D4	Chuva	Ameno	Elevada	Fraco	Sim
D5	Chuva	Fresco	Normal	Fraco	Sim
D6	Chuva	Fresco	Normal	Forte	Não
D7	Nuvens	Fresco	Normal	Fraco	Sim
D8	Sol	Ameno	Elevada	Fraco	Não
D9	Sol	Fresco	Normal	Fraco	Sim
D10	Chuva	Ameno	Normal	Forte	Sim
D11	Sol	Ameno	Normal	Forte	Sim
D12	Nuvens	Ameno	Elevada	Forte	Sim
D13	Nuvens	Quente	Normal	Fraco	Sim
D14	Chuva	Ameno	Elevada	Forte	Não

Primeiramente são analisados todos os atributos (Aspecto, Temp., Humidade, Vento), começando, por exemplo, pela Humidade:

$S = [9+, 5-]$

$$E = -\frac{9}{14} log_2 \frac{9}{14} - \frac{5}{14} log_2 \frac{5}{14} = 0.940$$



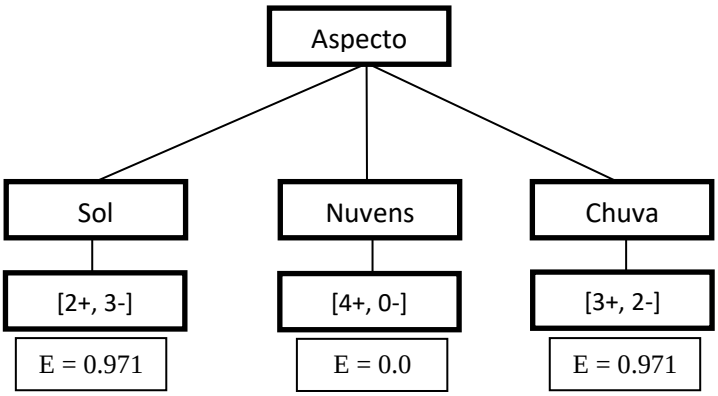
$$Ganho(S, Humidade) = 0.940 - \left(\frac{7}{14}\right) 0.985 - \left(\frac{7}{14}\right) 0.592 = 0.151$$

Repetindo-se o cálculo para os demais atributos é observado que o Aspecto possui o maior ganho:

$$Max \begin{pmatrix} Ganho(S, Humidade) = 0.151 \\ Ganho(S, Vento) = 0.048 \\ Ganho(S, Aspecto) = 0.247 \\ Ganho(S, Temp.) = 0.029 \end{pmatrix} = Ganho(S, Aspecto)$$

$S = [9+, 5-]$

$E = 0.940$



$$Ganho(S, Apecto) = 0.940 - \left(\frac{5}{14}\right)0.971 - \left(\frac{4}{14}\right)0.0 - \left(\frac{5}{14}\right)0.971 = 0.247$$

No próximo passo devemos desconsiderar o atributo escolhido (Aspecto) e aplicar o algoritmo recursivamente em seus filhos. Como exemplo, iremos calcular o ganho (redução esperada) na entropia de 'Sol', ordenado pelos atributos Humidade, Temp. e Vento:

$$Max\left(\begin{array}{c} Ganho(Sol, Humidade) = 0.971 - \left(\frac{3}{5}\right)0.0 - \left(\frac{2}{5}\right)0.0 = 0.971 \\ Ganho(Sol, Temp.) = 0.971 - \left(\frac{2}{5}\right)0.0 - \left(\frac{2}{5}\right)1.0 - \left(\frac{1}{5}\right)0.0 = 0.570 \\ Ganho(Sol, Vento) = 0.971 - \left(\frac{2}{5}\right)1.0 - \left(\frac{3}{5}\right)0.918 = 0.019 \end{array}\right) = Ganho(Sol, Humidade)$$

O algoritmo termina quando todos os nós possuírem entropia nula.

CFBioinfo: http://kdbio.inesc-id.pt/~atf/bioinformatics.ath.cx/bioinformatics.ath.cx/indexf23d.html?id=199

Referências:

[1] Borje Karlson. **Um Middleware de Inteligência Artificial para Jogos Digitais**. Dissertação de Mestrado, PUC-Rio, 2005.

[2] Pozzer, C. T. **Um Sistema para Geração, Interação e Visualização Tridimensional de Histórias para TV Interativa**. Tese de Doutorado, PUC-Rio, 2005.

[3] Pozzer, C. T., Feijo, B., Ciarlini, A., Furtado, A., Dreux, M. **Managing Actions and Movements of Non-Player Characters in Computer Games**. In: SBGames: Simpósio Brasileiro de Jogos de Computador e Entretenimento Digital, WJogos 2004, Curitiba, 2004.

[4] Matt Buckland. **Programming Game AI by Example**. Wordware publishing Inc, 2005.

[5] Gilliard Lopes. **Notas de Aula**. PUC-Rio, 2005.

[6] Ryan Houlette, Dan Fu. **Construction a Decision Tree Based on Past Experience**. In: AI Game programming wisdom 2. Charles River Media, 2004.

[7] Richard Evans. **Varieties of Learning**. In: AI Game programming wisdom. Charles River Media, 2002.

[8] **Tutorial: Machine Learning.** Disponível em: http://www.gamasutra.com/gdc2005/features/20050307/postcard-sanchez-crespo.htm

[9] Stuart Russel, Peter Norvig. **Artificial Intelligence, a modern Approach**. Second edition, Prentice Hall, 2003.