INSTITUTO DE ENSINO SUPERIOR DO ESPÍRITO SANTO
FACULDADE DO ESPÍRITO SANTO - MULTIVIX CACHOEIRO DE ITAPEMIRIM
CURSO DE SISTEMAS DE INFORMAÇÃO

MARCELO CONTARINI DE SOUZA

A UTILIDADE DO ALGORITMO MINIMAX EM JOGOS DIGITAIS

CACHOEIRO DE ITAPEMIRIM
2014

MARCELO CONTARINI DE SOUZA

A UTILIDADE DO ALGORITMO MINIMAX EM JOGOS DIGITAIS

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação na Faculdade do Espírito Santo – Multivix Cachoeiro de Itapemirim como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Profº. Orientador: Me. Ricardo Maroquio Bernardo

CACHOEIRO DE ITAPEMIRIM
2014

MARCELO CONTARINI DE SOUZA

A UTILIDADE DO ALGORITMO MINIMAX EM JOGOS DIGITAIS

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação na Faculdade do Espírito Santo - Multivix Cachoeiro de Itapemirim, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Aprovado em 03 de dezembro de 2014.

COMISSÃO EXAMINADORA

_____

Profº. Orientador: Me. Ricardo Maroquio Bernardo

_____

Profº. Convidado: Esp. Cleziel Franzoni da Costa

_____

Profº. Convidado: Esp. Bernardo Casotti Vidaurre Ávilla Paldês

Dedico a Deus por ter me dado forças para a conclusão deste trabalho.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado coragem de resolver todos os problemas que foram aparecendo na minha vida, à minha família por todo tipo de apoio recebido durante o curso de graduação, aos meus amigos ao longo desta caminhada. Aos meus professores, principalmente ao meu orientador, por ter me passado seu conhecimento.

“O mundo não está preocupado com a vossa autoestima. O mundo espera que vocês façam alguma coisa útil por ele antes de vocês se sentirem bem convosco próprios.”
Bill Gates

SOUZA, Marcelo Contarini. **A Utilidade do Algoritmo Minimax em Jogos Digitais.** 2014. Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação) – Faculdade do Espirito Santo – Multivix Cachoeiro de Itapemirim, Cachoeiro de Itapemirim, 2014.

RESUMO

O grande progresso no campo de hardware ao passar das décadas, permitiu também que o campo de software evoluísse. Nesta evolução do campo de software, está sendo levado em consideração, especificamente, os algoritmos aplicados neles. O campo da Inteligência Artificial tem como um dos principais motivos resolver problemas complexos, e, por isso, os algoritmos do campo têm um custo computacional muito alto, mas o progresso no campo de hardware compensou esse custo. Este trabalho aborda o Minimax, algoritmo de Inteligência Artificial que é aplicado em Jogos Digitais para resolver o problema de decisão de jogada efetuada pelo jogador artificial. O Minimax é um algoritmo recursivo e cria uma árvore com todas as possibilidades de jogadas, e com base em uma heurística, é decidida a melhor jogada para o jogador artificial. Jogos em que o Minimax é aplicado possuem características peculiares, como dois jogadores, competitivamente alternando entre turnos, sendo que cada jogador sempre possui informações do estado do jogo, como por exemplo, no jogo da velha, damas e xadrez. Neste trabalho, o algoritmo Minimax será aplicado no jogo de cartas Triple Triad (como estudo de caso), para criar um jogador artificial em que é capaz de fazer boas jogadas para vencer o adversário. O jogo original foi desenvolvido pela Squaresoft em 1999.

Palavras-chave: Algoritmos. Inteligência Artificial. Minimax. Jogos Digitais.

SOUZA, Marcelo Contarini. **A Utilidade do Algoritmo Minimax em Jogos Digitais.** 2014. Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação) – Faculdade do Espirito Santo – Multivix Cachoeiro de Itapemirim, Cachoeiro de Itapemirim, 2014.

ABSTRACT

The great progress in the field of hardware over the decades, also allowed the field of software evolve. On this evolution of the field of software, is being taken into consideration, specifically, the algorithms applied on them. The field of Artificial Intelligence has as main goals to solve complex problems, and, therefore, the algorithms of the field have a very high computational cost, but the progress in the field of hardware offsets this cost. This work addresses the Minimax, Artificial Intelligence algorithm that is applied in Digital Games, to solve the problem of move decision made by the artificial player. The Minimax is a recursive algorithm and creates a tree with all possible moves, and based on a heuristic, is decided the best move for the artificial player. Games in which Minimax is applied have peculiar characteristics, as two players, competitively alternating turns, and each player always has information of the state of the game, for example in tic-tac-toe, checkers and chess. In this work, the Minimax algorithm will be applied to the card game Triple Triad (as a case study), to create an artificial player that is able to make good moves to beat the opponent. The original game was developed by Squaresoft in 1999.

Key words: Algorithms. Artificial Intelligence. Minimax. Digital Games.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – Application Programming Interface

GPS – General Problem Solver

VCS – Video Computer System

NES – Nintendo Entertainment System

RPG – Role-Playing Game

SUMÁRIO

1 INTRODUÇÃO

O hardware do computador é uma coleção das partes físicas de um sistema. Isto inclui o gabinete do computador, monitor, teclado e mouse. Também inclui todas as peças dentro do gabinete do computador, como a unidade de disco rígido, placa-mãe, placa de vídeo, e muitos outros. Basicamente é o que é possível tocar fisicamente. Os componentes básicos de um computador pessoal hoje são mais ou menos o mesmo como eram a algumas décadas atrás. As peças ainda executam as mesmas funções gerais como faziam. A placa-mãe ainda serve como ponto central do computador, com tudo o que se conectar a ela, o processador ainda executa instruções, a memória ainda armazena dados para um acesso rápido e discos rígidos ainda armazenam dados de longo prazo. A forma como essas peças são conectadas e a rapidez com que operam tem mudado muito. Quanto mais circuitos integrados ou transistores um chip tem, mais rápido ele vai ser. O hardware evoluiu muito (e ainda evolui) nas últimas décadas, aumentando o número de possibilidades de sua utilização, mas acaba sendo inutilizado sem um componente importante: o software.

O software é um programa que permite ao computador executar uma tarefa específica, ao contrário dos componentes físicos do sistema (hardware). Isto inclui software de aplicação, tais como um processador de texto, que permite que um usuário execute uma tarefa, e software de sistema, como um sistema operacional, que permite que outro software seja executado corretamente, por interface com o hardware e outro software. O software depende do hardware para funcionar corretamente e deve ser carregado na memória do computador. Uma vez que o software é carregado, o computador é capaz de executá-lo. Trata-se de passar as instruções da aplicação, através do sistema operacional, para o hardware, que finalmente recebe a instrução de código de máquina, ou seja, o código que a máquina entenda. Cada instrução faz com que o computador realize uma operação: movimentação de dados, realização de um cálculo, ou alteração do fluxo de instruções de controle. O software é desenvolvido em uma linguagem de programação (como C, C++, C# e Java) para que seja executada uma sequência de instruções feita pelo programador.

Como o software é necessário para tratar de questões de automação, de adaptação, de otimização e escalabilidade, outras áreas da computação são necessárias. A Inteligência Artificial é uma das áreas que podem levar o software ainda mais longe. Por outro lado, as técnicas da Engenharia de Software também podem desempenhar um papel importante para aliviar o custo de desenvolvimento e tempo das técnicas de Inteligência Artificial, bem como auxiliar na introdução de novas técnicas. Tais benefícios mútuos têm aparecido nas últimas décadas e ainda evoluiu devido a novos desafios.

Com a grande mudança que os computadores estão fazendo em nossas vidas, as técnicas de Inteligência Artificial estão pavimentando o caminho para esta transformação rápida. Os algoritmos (que são um conjunto de regras a serem seguidas em cálculos, para resolução de problemas, executadas por um computador) de Inteligência Artificial permitem a construção de sistemas inteligentes que podem operar de forma autônoma, aprender com a experiência, planejar suas ações e resolver problemas complexos. Os algoritmos de Inteligência Artificial têm um custo computacional muito grande, e a evolução do hardware ao longo das décadas contribuiu muito para que estes algoritmos sejam aplicados nos softwares utilizados nos dias de hoje. As aplicações incluem robôs que planejam suas próprias ações, rastreadores da web que localizam informações de forma eficiente, assistentes inteligentes que ajudam os seres humanos detectar fraudes financeiras e jogos competitivos com jogadas que são melhores do que qualquer jogador humano possa executar.

Dentre vários algoritmos de Inteligência Artificial utilizados em Jogos Digitais, o algoritmo Minimax é utilizado em jogos de mesa (ou tabuleiro), como, por exemplo, jogo da velha e xadrez, onde 2 jogadores jogam pela vitória, ou seja, o jogador luta pela a vitória sem cooperação do outro. O algoritmo faz uma busca em uma estrutura de dados com todas as jogadas possíveis, e sempre optando pelo caminho em que vai trazer maiores chances de vitória para o jogador (maximizar) e ao mesmo tempo, maiores chances de derrota para o adversário (minimizar). Sendo assim, com o algoritmo aplicado em um jogo, é possível torná-lo mais desafiador caso o jogador humano queira jogar contra o jogador artificial.

Neste trabalho o algoritmo Minimax será aplicado no jogo Triple Triad, para criar um jogador artificial que seja capaz de vencer o adversário.

1.1 Objetivos

Como objetivo geral, o presente trabalho visa aplicar o algoritmo Minimax (algoritmo de Inteligência Artificial) para resolver o problema da decisão de jogada no jogo Triple Triad (jogo de cartas) para a plataforma PC.

Para alcançar o objetivo geral, este trabalho tem como objetivos específicos o seguinte:

a) Abordar pesquisas e análises de tipos de projetos que utilizam o algoritmo Minimax;
b) Mostrar como fazer um jogo da velha para aplicar o algoritmo Minimax;
c) Apresentar a implementação do jogo Triple Triad para 2 jogadores humanos;
d) Observar o comportamento das jogadas e decisões realizadas pelo computador no jogo original do Triple Triad;
e) Implementar o jogo Triple Triad com o Minimax aplicado para jogar contra o jogador artificial.

1.2 Justificativa

A justifica deste trabalho é a contribuição acadêmica demonstrando como aplicar o algoritmo Minimax em um jogo (incluindo o código fonte), e desta forma, trazendo o desafio para o jogador: com o algoritmo Minimax aplicado, o jogo se tornar mais interessante e desafiador para o jogador, trazendo a satisfação e diversão do jogo.

1.3 Metodologia

Para o desenvolvimento deste projeto, foi realizada uma revisão teórica/bibliografia, trabalhos acadêmicos e outros materiais presentes em páginas da Web.

Foi desenvolvido um framework para jogos, para a implementação do jogo Triple Triad e a aplicação do algoritmo Minimax no jogo. Foi utilizado o software Microsoft

Visual Studio 2013 Ultimate e o projeto foi todo desenvolvido utilizando a linguagem de programação C++.

O framework criado utiliza as seguintes bibliotecas e/ou APIs (*Application Programming Interface*, traduzindo, Interface de Programação da Aplicação):

DirectX, utilizando as APIs: Direct3D, XAudio2, D3DCompiler, DirectInput (MICROSOFT, Acesso em: 08 agosto 2014).
FreeImage, (FREEIMAGE, Acesso em: 08 agosto 2014).
FreeType, (FREETYPE, Acesso em: 08 agosto 2014).
Ogg Vorbis, (VORBIS, Acesso em: 08 agosto 2014).
OpenAL, (OPENAL, Acesso em: 08 agosto 2014).

1.4 Organização do Trabalho

Após a introdução feita no presente capítulo, o Capítulo 2 aborda a evolução e importância da área da Inteligência Artificial, como Jogos Digitais surgiram e evoluíram, e também aborda sobre a estrutura de dados do tipo árvore.

Em seguida, o Capítulo 3 aborda sobre o algoritmo Minimax, como suas características, em quais tipos de jogos pode-se aplicá-lo e também próprio algoritmo. Também aborda a importância da função de utilidade do algoritmo Minimax e sobre o corte alfa-beta, para otimizar o algoritmo Minimax.

No Capítulo 4 aborda sobre o jogo de cartas Triple Triad, como informações gerais, regras, funcionamento, e também o código fonte com o algoritmo Minimax aplicado e comentado. Também aborda o código fonte do algoritmo Minimax aplicado e comentado em um jogo da velha.

Por fim, o Capítulo 5 apresenta a conclusão que pôde ser tirada mediante das experiências obtidas nos capítulos anteriores.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, será apresentado uma visão geral sobre a evolução da Inteligência Artificial, como as contribuições feitas pelos pesquisadores ao longo dos anos, tem ajudado a resolver problemas complexos, sobre a história dos Jogos Digitais e sua evolução e também sobre a estrutura de dados do tipo árvore.

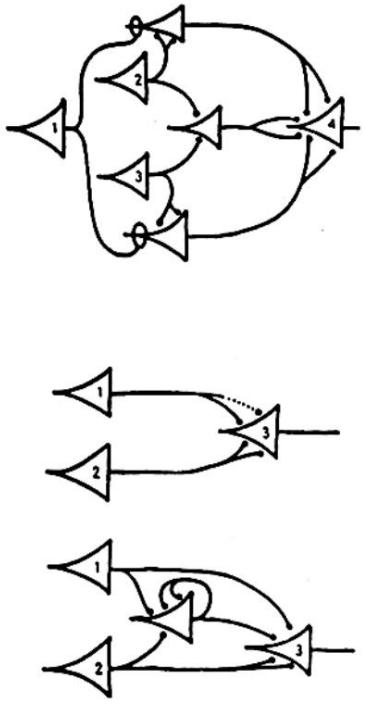
2.1 Inteligência Artificial

De acordo com Warwick (2011) o campo da Inteligência Artificial realmente veio à existência com o nascimento dos computadores e em torno de 1940 e 1950. No período inicial de seu desenvolvimento, a atenção foi claramente focada em conseguir fazer com que os computadores fizessem coisas que, se um ser humano fizesse, seria considerado como inteligente. Essencialmente, isto envolveu em tentar fazer com que os computadores copiassem os seres humanos em alguns ou todos os aspectos de seus comportamentos.

Segundo Norvig e Russell (2009), o primeiro trabalho, que hoje é geralmente reconhecido como Inteligência Artificial, foi feito por Warren McCulloch e Walter Pitts em 1943. Eles se basearam em três fontes: o conhecimento da fisiologia básica e função dos neurônios no cérebro, em uma análise formal da lógica proposicional de Russell e Whitehead e na teoria da computação de Turing. Eles propuseram um modelo de neurônios artificiais em que cada neurônio é caracterizado como sendo “ligado” ou “desligado”, com um interruptor para “ligado” que ocorre em resposta à estimulação por um número suficiente de neurônios vizinhos. O estado de um neurônio foi concebido como "factualmente equivalente a uma proposição que propôs o seu estímulo adequado." Eles mostraram, por exemplo, que qualquer função computável poderia ser computada por alguma rede neural conectada, e que todos os conectivos lógicos (e, ou, não, etc) poderiam ser implementados por simples estruturas de rede. McCulloch e Pitts também sugeriram que as redes definidas adequadamente poderiam aprender. Donald Hebb em 1949 demonstrou uma simples regra de atualização para modificar as forças de conexão entre os neurônios. Sua regra, agora chamada de aprendizagem Hebbiana, continua a ser um modelo influente até hoje.

Já Nilsson (2009) afirma que McCulloch e Pitts alegaram que o neurônio era, em essência, uma "unidade lógica." Em um famoso e importante artigo que eles propuseram modelos simples de neurônios e mostraram que essas redes destes modelos poderiam realizar todas as possíveis operações computacionais. O “neurônio” de McCulloch e Pitts era uma abstração matemática com entradas e saídas (o que corresponde, aproximadamente, a dendritos e axônios, respectivamente). Cada um pode ter o valor 1 ou 0. Estes neurônios podem ser conectados juntos em rede de tal forma que a saída de um neurônio é uma entrada para os outros e assim por diante. Alguns neurônios são excitatórios: os seus resultados contribuem para "disparar" os neurônios aos quais estão ligados. Outros são inibitórios – suas saídas contribuem para inibir o disparo dos neurônios aos quais eles estão ligados. Se a soma das entradas dos neurônios excitatórios, menos a soma das entradas dos neurônios inibitório que incidem em um neurônio é maior do que um determinado valor “*threshold*” (limiar, ou seja, valor limite para uma reação ou resultado), aquele neurônio dispara, enviando a sua saída de 1 para todos os neurônios aos quais está ligado. Alguns exemplos de redes propostos por McCulloch e Pitts são mostrados na figura 01:

Figura 01 – Elementos da rede neural de McCulloch e Pitts



Fonte: Nilsson (2009)

De acordo com Norvig e Russell (2009), dois estudantes de graduação de Harvard, Marvin Minsky e Dean Edmonds construiram o primeiro computador de rede neural em 1950. O SNARC, como foi chamado, usava 3.000 tubos de vácuo e um excedente mecanismo de piloto automático de um bombardeiro B-24 para simular uma rede de 40 neurônios. Mais tarde, em Princeton, Minsky estudou computação universal em redes neurais. O comitê do seu Ph.D. estava cético sobre se este tipo de trabalho deveria ser considerado matemática, mas von Neumann teria dito: "Se não for agora, será um dia." Minsky mais tarde provou teoremas influentes mostrando as limitações das pesquisas em rede neural.

Segundo Norvig e Russell (2009), houve vários exemplos de trabalhos no início que poderiam ser caracterizados como Inteligência Artificial, mas a visão de Alan Turing foi talvez a mais influente. Ele deu palestras sobre o tema já em 1947, na Sociedade de Matemática de Londres e articulou uma agenda persuasiva em seu artigo de 1950, intitulado "*Computing Machinery and Intelligence*." Nisso, ele introduziu o Teste de Turing, aprendizagem de máquina, algoritmos genéticos, e aprendizado por reforço. Ele propôs a ideia do "*Child Programme*" explicando "Em vez de tentar produzir um programa para simular a mente adulta, por que não tentar produzir uma que simulava a mente de uma criança?". Já Nilsson (2009) afirma que o primeiro artigo moderno a lidar com a possibilidade de mecanizar toda a inteligência de estilo-humano foi publicado por Turing em 1950. Este artigo é famoso por várias razões. Turing pensou que a pergunta "uma máquina pode pensar?" era muito ambígua. Em vez disso, ele propôs que a questão da inteligência da máquina fosse resolvida por aquilo que veio a ser chamado de Teste de Turing. No Teste de Turing, ao invés de tentar determinar se uma máquina está pensando, Turing sugere que deveríamos perguntar se a máquina pode ganhar em um jogo, chamado de "*Imitation Game*". Trata-se de três participantes em salas isoladas: um computador (que está sendo testado), um ser humano e um juiz (humano). O juiz pode conversar tanto com o ser humano quanto com o computador, digitando em um terminal. Tanto o computador e humano tentam convencer o juiz de que eles são um humano. Se o juiz não disser qual é qual, em seguida, o computador ganha o jogo.

De acordo com Norvig e Russell (2009), Princeton foi a casa de uma outra figura influente na Inteligência Artificial, John McCarthy. Depois de receber seu doutorado lá em 1951 e ter trabalhado por dois anos como instrutor, McCarthy se mudou para Stanford e depois para Dartmouth College, que viria a ser o local de nascimento oficial do campo. McCarthy convenceu Minsky, Claude Shannon, e Nathaniel Rochester para ajudá-lo a reunir pesquisadores norte-americanos interessados em teoria dos autômatos, redes neurais, bem como no estudo da inteligência. Eles organizaram uma oficina de dois meses, em Dartmouth, no verão de 1956.

Segundo Norvig e Russell (2009) e Nilsson (2009) a proposta da oficina dizia que: “Propomos que em dois meses, um estudo de Inteligência Artificial com 10 homens seja realizado durante o verão de 1956 no Dartmouth College, em Hanover, New Hampshire. O estudo é para prosseguir com base na conjectura de que cada aspecto de aprendizado ou qualquer outro recurso de inteligência pode, em princípio, ser descrito com tanta precisão que uma máquina pode ser feita para simulá-lo. Será feita uma tentativa de encontrar como fazer máquinas para usar a linguagem, formar abstrações e conceitos, resolver os tipos de problemas agora reservados para os seres humanos, e melhorar a si mesmos. Pensamos que um avanço significativo pode ser feito em um ou mais desses problemas, se um grupo cuidadosamente selecionado de cientistas trabalhar em conjunto em um verão.”

De acordo com Norvig e Russell (2009), a oficina de Dartmouth não levou a quaisquer novas descobertas, mas fez com que todos os grandes nomes se apresentassem entre eles. Para os próximos 20 anos, o campo seria dominado por essas pessoas e seus alunos e colegas do MIT, CMU, Stanford e IBM. Olhando para a proposta da oficina Dartmouth, podemos ver por que era necessário a Inteligência Artificial se tornar um campo separado. Foram levantadas algumas questões como: por que não poderia todo o trabalho feito em Inteligência Artificial ter um lugar sob o nome de teoria de controle ou pesquisa de operações ou teoria de decisão, que afinal, têm objetivos semelhantes aos da Inteligência Artificial? Ou porque não a Inteligência Artificial ser um ramo da Matemática? A primeira resposta é que a Inteligência Artificial desde o início abraçou a ideia de duplicar faculdades humanas como criatividade, auto aperfeiçoamento e uso da língua. Nenhum dos outros campos estavam tratando estas questões. A segunda resposta é a metodologia. A

Inteligência Artificial é o único destes campos que é claramente um ramo da ciência da computação (embora a pesquisa de operações compartilha uma ênfase em simulações de computador) e a Inteligência Artificial é o único campo que tenta construir máquinas que funcionarão de forma autônoma em complexos ambientes em mudança.

Segundo Warwick (2011) na década de 1960 a mais profunda contribuição para o campo foi sem dúvida o GPS (*General Problem Solver*, traduzindo, Solucionador de Problemas Gerais) de Newell e Simon. Este foi um programa de propósito múltiplo que visava simular, usando um computador, alguns métodos de resolução de problemas humanos. De acordo com Norvig e Russell (2009), dentro de limitados tipos de quebra-cabeças que poderia lidar, descobriu-se que a ordem em que o programa considerava submetas e possíveis ações eram semelhante ao que, os seres humanos se aproximavam dos mesmos problemas. Assim, GPS foi provavelmente o primeiro programa a incorporar a abordagem de "pensar humanamente". Warwick (2011) afirma que infelizmente, a técnica empregada não foi particularmente eficiente, e por causa do tempo necessário e os requisitos de memória para resolver até mesmo problemas reais relativamente simples, por isso o projeto foi abandonado.

De acordo com Norvig e Russell (2009) na IBM, Nathaniel Rochester e seus colegas produziram alguns dos primeiros programas de Inteligência Artificial. Herbert Gelernter (1959) construiu o *Geometry Theorem Prover*, que foi capaz de provar teoremas que muitos estudantes de matemática achariam bastante complicado. A partir de 1952, Arthur Samuel escreveu uma série de programas para damas (rascunhos) que finalmente aprendeu a jogar em um nível forte amador. Ao longo do caminho, ele descartou a ideia de que os computadores podem fazer apenas o que lhes é dito para: seu programa rapidamente aprendeu a jogar um jogo melhor do que o seu criador. O programa foi demonstrado na televisão, em fevereiro de 1956, criando uma forte impressão.

Segundo Norvig e Russell (2009) John McCarthy se mudou de Dartmouth para o MIT e lá fez algumas contribuições cruciais em um ano histórico: 1958. No MIT AI Lab Memo No. 1, McCarthy definiu a linguagem de alto nível LISP, o que viria a ser

a linguagem dominante de programação de Inteligência Artificial para os próximos 30 anos. Publicou um artigo intitulado “*Programs with Common Sense*”, no qual ele descreveu o *Advice Taker* (traduzindo, Tomador de Conselhos), um programa hipotético que pode ser visto como o primeiro sistema de Inteligência Artificial completo. De acordo com Nilsson (2009), era representado como expressões em uma linguagem matemática (e favorável ao computador) chamada de "lógica de primeira ordem." Por exemplo, os fatos "Estou na minha mesa" e "Minha mesa está em casa" poderiam ser representados como as expressões em (eu, mesa) e em (mesa, casa).

Norvig e Russell (2009) afirmam que Minsky supervisionou uma série de alunos que escolheram problemas limitados que pareciam exigir inteligência para resolver. Estes domínios limitados ficaram conhecido como microworlds. O programa SAINT de James Slagle em 1963, foi capaz de resolver de forma fechada problemas de cálculo integral típicos de primeiro ano de cursos universitários. O programa ANALOGY de Tom Evans em 1968, resolveu problemas de analogia geométricas que aparecem nos testes de Quociente de Inteligência. O programa STUDENT de Daniel Bobrow em 1967 resolveu problemas de álgebra com histórias.

De acordo com Norvig e Russell (2009) o quadro de resolução de problemas que surgiram durante a primeira década de pesquisas em Inteligência Artificial eram de um mecanismo de busca de propósito geral tentando encadear passos de raciocínio fundamental para encontrar soluções completas. Essas abordagens têm sido chamadas de métodos fracos, porque não ampliavam até instâncias de problemas grandes ou difíceis. A alternativa aos métodos fracos é usar um conhecimento de domínio específico mais poderoso que permite passos de raciocínio maiores e que podem facilmente lidar com casos que ocorrem normalmente em áreas estreitas de especialização. Pode-se dizer que para resolver um problema difícil, é quase necessário já saber a resposta.

Segundo Norvig e Russell (2009) o programa DENDRAL foi um dos primeiros exemplos dessa abordagem. Ele foi desenvolvido na Universidade de Stanford, onde Ed Feigenbaum (um ex-aluno de Herbert Simon), Bruce Buchanan (um filósofo que virou cientista da computação), e Joshua Lederberg (um laureado com o Nobel

geneticista) uniram-se para resolver o problema de inferir a estrutura molecular a partir da informação fornecida por um espectrómetro de massa. A importância do programa DENDRAL é que era o primeiro sistema intensivo de conhecimento bem sucedido: a sua experiência derivada de um grande número de regras de propósito específico. Sistemas posteriores também incorporaram o tema principal da abordagem do *Advice Taker* de McCarthy: a separação limpa do conhecimento (sob a forma de regras) a partir do componente de raciocínio.

Norvig e Russell (2009) afirmam que, com essa lição em mente, Feigenbaum e outros em Stanford começaram o *Heuristic Programming Project* (traduzindo, Projeto de Programação Heurística) para investigar até que ponto a nova metodologia de sistemas especialistas poderia ser aplicada a outras áreas do conhecimento humano.

Segundo Warwick (2011), a década de 1980 presenciou uma espécie de renascimento em Inteligência Artificial. Muitos pesquisadores seguiram o exemplo de McCarthy e continuaram a desenvolver sistemas de Inteligência Artificial a partir de um ponto de vista prático. Este período viu o desenvolvimento de sistemas especialistas, que foram projetados para lidar com um domínio muito específico de conhecimento, daí evitando um pouco os argumentos com base na falta de "senso comum". Embora inicialmente testado na década de 1970, foi na década de 1980 que tais sistemas começaram a ser usados para reais aplicações práticas na indústria.

De acordo com Norvig e Russell (2009) o primeiro sistema especialista comercialmente bem sucedido foi o R1, que começou sua operação na Digital Equipment Corporation em 1982. O programa ajudou a configurar encomendas de novos sistemas e por volta de 1986 estava economizando para a empresa, cerca de 40 milhões de dólares por ano. Em 1988, o setor de Inteligência Artificial da Digital Equipment Corporation tinha 40 sistemas especialistas implantados e com mais alguns que estavam sendo desenvolvidos. DuPont teve 100 sistemas em uso e 500 em desenvolvimento, economizando cerca de 10 milhões de dólares por ano. Quase todas as grandes corporações dos EUA tinham seu próprio setor de Inteligência Artificial para usar ou investigar sistemas especialistas. No geral, a indústria de

Inteligência Artificial cresceu de alguns milhões de dólares em 1980 para bilhões de dólares em 1988, incluindo centenas de empresas de construção de sistemas especialistas, sistemas de visão, robôs, software e hardware especializados para esses fins.

Segundo Norvig e Russell (2009) em meados da década de 1980, pelo menos quatro grupos diferentes reinventaram o algoritmo de aprendizado *back-propagation* encontrado pela primeira vez em 1969 por Bryson e Ho. O algoritmo foi aplicado a muitos problemas de aprendizagem em ciência da computação e psicologia.

Norvig e Russell (2009) afirmam que de 1987 até hoje houve uma revolução, tanto no conteúdo e na metodologia de trabalho em Inteligência Artificial. Agora é mais comum construir trabalhos sobre teorias existentes do que propor novos, para basear-se em alegações sobre teoremas rigorosos ou de difícil evidência experimental do que a intuição, e para mostrar relevância para aplicações do mundo real, em vez de exemplos irreais. A Inteligência Artificial foi fundada em parte como uma rebelião contra as limitações de campos já existentes, como a teoria de controle e estatísticas, mas agora ela está abraçando estes campos. Em termos de metodologia, a Inteligência Artificial finalmente chegou firmemente sob o método científico. Para ser aceito, as hipóteses devem ser submetidas a experimentos empíricos rigorosos, e os resultados devem ser analisados estatisticamente para a sua importância. Agora é possível replicar experimentos usando repositórios compartilhados de dados e código de teste.

De acordo com Norvig e Russell (2009), o campo de reconhecimento de voz representa este método. Na década de 1970, uma grande variedade de diferentes arquiteturas e abordagens foram tentadas. Muitas delas eram feitas para um propósito particulares e eram frágeis, por isso foram demonstradas em apenas alguns exemplos especialmente selecionados. Nos últimos anos, as abordagens baseadas em *hidden Markov models* (traduzindo, modelos ocultos de Markov) passaram a dominar a área. Dois aspectos são relevantes. Em primeiro lugar, elas são baseadas em uma teoria matemática rigorosa. Isto permitiu aos pesquisadores desenvolver sobre resultados matemáticos de várias décadas, desenvolvidos em outras áreas. Em segundo lugar, elas são geradas por um processo de formação de

uma grande quantidade de dados de voz real. Isso garante que o desempenho é robusto, e em rigorosos testes cegos os modelos ocultos de Markov têm melhorado a sua pontuação de forma constante.

Norvig e Russell (2009) afirmam que a tradução automática segue o mesmo curso que o reconhecimento de voz. Na década de 1950 houve entusiasmo inicial para uma abordagem baseada em sequências de palavras, com os modelos aprendidos de acordo com os princípios da teoria da informação. Essa abordagem caiu em desuso na década de 1960, mas voltou no final de 1990 e agora domina o campo. De acordo com Nilsson (2009) algumas das primeiras tentativas de usar computadores para mais do que os cálculos numéricos usuais estavam em tradução automática de sentenças em uma língua em sentenças de outra. Dicionários de palavras podem ser armazenados na memória do computador, e estes poderiam ser usados para encontrar palavras em um idioma para palavra equivalentes em outros idiomas. Pensava-se que escolhendo uma adequada palavra equivalente para cada palavra estrangeira em uma frase, juntamente com uma quantidade modesta de análise sintática, poderia-se traduzir uma frase em uma língua estrangeira (russo, por exemplo) para o inglês.

Segundo Norvig e Russell (2009), as redes neurais também se encaixam nessa tendência. Grande parte do trabalho em redes neurais na década de 1980 foi feito na tentativa de verificar o que poderia ser feito e para saber como redes neurais diferem das técnicas "tradicionais". Utilizando metodologias melhoradas e estruturas teóricas, o campo chegou a um entendimento em que as redes neurais podem agora ser comparadas com as técnicas correspondentes de estatísticas, reconhecimento de padrões e aprendizagem de máquina, e a técnica mais promissora pode ser aplicada caso a caso. Como resultado destes desenvolvimentos, a chamada tecnologia de *Data Mining* (traduzindo, mineração de dados) gerou uma nova indústria vigorosa.

De acordo com Norvig e Russell (2009) o formalismo da rede Bayesiana foi inventado para permitir a representação eficiente de raciocínio rigoroso com o conhecimento incerto. Esta abordagem supera largamente muitos problemas dos sistemas de raciocínio probabilístico dos anos 1960 e 1970. Agora domina a

pesquisa na Inteligência Artificial em sistemas especialistas e de raciocínio incerto. A abordagem permite aprender com a experiência, e que combina o melhor de redes neurais e Inteligência Artificial clássica.

Segundo Norvig e Russell (2009), por volta de 1995 surgiram os agentes inteligentes. O trabalho de Allen Newell, John Laird, e Paul Rosenbloom é o exemplo mais conhecido de uma arquitetura de agente completa. Um dos ambientes mais importantes para agentes inteligentes é a Internet. Sistemas de Inteligência Artificial se tornaram tão comuns em aplicativos baseados na Web que o sufixo "-bot" entrou na linguagem cotidiana. Além disso, as tecnologias de Inteligência Artificial estão por trás de muitas ferramentas da Internet, tais como motores de busca, sistemas de recomendação e agregadores de sites.

A Inteligência Artificial é um campo muito bem definido, que tem crescido ao longo dos últimos 50 anos e tem ajudado a resolver muitos problemas. Hoje utilizamos muitas destas tecnologias criadas por esses pesquisadores que contribuíram muito para facilitar nossas vidas. Dentro deste campo existem muitas áreas que ainda vão evoluir. Algoritmos de Inteligência Artificial podem ser aplicados na área de Jogos Digitais, como por exemplo, para simular a inteligência de um jogador adversário.

2.2 Jogos Digitais

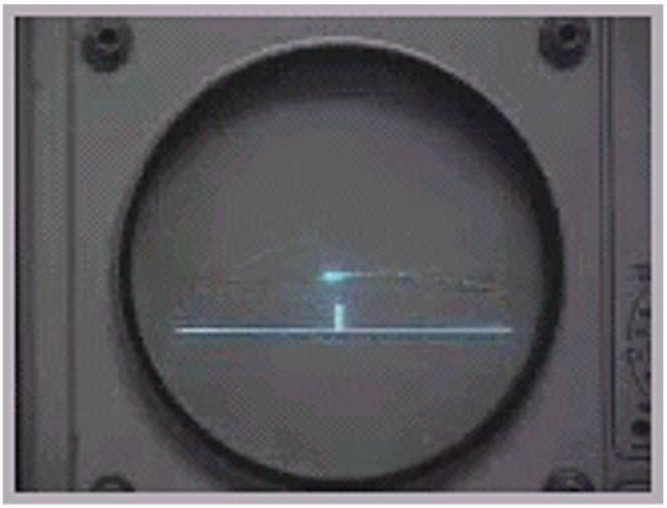
Segundo Donovan (2010), no início de 1958, o vídeo game ainda era um conceito evasivo. Os cientistas da computação ainda viam jogos como folhas para suas pesquisas e os engenheiros viam potencial na TV para ser uma experiência bidirecional entre a tela e o espectador, mas falharam em desenvolver as suas ideias. O Nimrod, de John M. Bennett (Primeiro computador digital de propósito específico projetado para jogar) ainda era a coisa mais próxima de um vídeo game que qualquer pessoa fora das oficinas de engenharia ou laboratório de informática da universidade tinha visto. Mas em 1958 o conceito de vídeo game surgiu, graças a um passo de William Higinbotham.

De acordo com Donovan (2010), Higinbotham havia trabalhado no Projeto Manhattan, construindo os interruptores temporais que faziam a bomba explodir no

momento correto. Como muitos dos cientistas que criaram a bomba, ele nutria sentimentos mistos sobre o que tinha feito e passava grande parte de sua campanha de vida pós-guerra contra a proliferação nuclear. Depois da guerra, tornou-se chefe da divisão de instrumentação no Laboratório Nacional de Brookhaven - um centro de pesquisa do governo dos Estados Unidos da América com base em Long Island, Nova York. Todos os anos Brookhaven abria suas portas ao público para mostrar trabalhos. Estes visitantes diários tendiam a conter exposições estáticas que pouco animavam o público e, portanto, com o dia aberto em 1958 se aproximando, Higinbotham decidiu fazer uma atração mais envolvente.

Donovan (2010) afirma que ele veio com a ideia de uma divertida exposição interativa: um jogo de tênis jogando na tela de um osciloscópio que construiu usando circuitos de transistores com a ajuda do engenheiro Brookhaven Robert Dvorak. O jogo, Tennis for Two (Tênis para Dois) conforme a figura 02, recriou uma visão lateral de um campo de tênis com uma rede no meio e linhas finas que representavam raquetes dos jogadores.

Figura 02 - Osciloscópio com o jogo Tennis for Two



Fonte: Lanzi (Acesso em: 05 agosto 2014)

Os grandes controladores em forma de caixa criados para o jogo permitiram que os jogadores movessem suas raquetes usando uma conexão dial e bater na bola, pressionando um botão. Os visitantes do Brookhaven adoraram isto. “Os estudantes do ensino médio gostavam mais, você não poderia tirá-los", lembrou Higinbotham mais de 20 anos depois. Na verdade Tennis for Two foi tão popular que ele retornou

para uma segunda aparição em 1959 no dia aberto em Brookhaven. Mas nem Higinbotham nem ninguém no Brookhaven tinha pensado muito no jogo e após isto, seu equipamento foi desmontado para que suas peças pudessem ser utilizadas em outros projetos.

De acordo com Kent (2001), alguns historiadores argumentam que Higinbotham, realmente inventou o primeiro jogo. Enquanto isso parece ser o primeiro jogo interativo, é um exemplo isolado. Aparentemente, nem Steven Russell nem Ralph Baer tinham conhecimento da existência do jogo de Higinbotham.

Três anos depois, em 1961, segundo Kushner (2004), Steve "Slug" Russell e um grupo de outros estudantes do Instituto de Tecnologia de Massachusetts criou o Spacewar! no primeiro minicomputador, o PDP-1 conforme a figura 03. Neste jogo, dois jogadores foguetes, enquanto flutuando em torno de um buraco negro. Berens e Howard (2008) afirmam que o Spacewar! introduziu conceitos que ainda são utilizados hoje: opções no jogo, um modo para dois jogadores, um sistema de pontuação e recursos limitados (neste caso mísseis e combustível). Projetado como uma forma divertida de mostrar o mais recente computador mainframe com um preço de mais de US$ 100.000, o jogo nunca foi patenteado e continuou a ser imitado por décadas, atuando como modelo até mesmo para os jogos posteriores como Asteroids. Foi também o início lento do aumento do número de vídeos games.

Figura 03 – Grupo de Russel criando o Spacewar! no PDP-1

Fonte: Lanzi (Acesso em: 05 agosto 2014)

Segundo Berens e Howard (2008) Nolan Bushnell, inspirado por seu trabalho de verão como gerente de uma galeria de pinball, onde os alunos do Instituto de Tecnologia de Massachusetts passavam para jogar Spacewar!, ao invés disso, ele sonhava com máquinas de jogos que funcionavam com moedas. Assim começou sua missão de produzir uma máquina compacta e financeiramente viável o suficiente para o trabalho.

De acordo com Berens e Howard (2008), Bushnell, com a ajuda de Ted Dabney, programador e fabricante da máquina de arcade na Nutting Associates, teve o melhor momento com a produção em massa do primeiro vídeo game que funcionava com moedas. Veio na forma de Computer Space em 1971, conforme a figura 04, em um jogo prontamente disponível, com circuitos lógicos, fios, reduzindo consideravelmente a escala e os custos da construção. (Reconhecimento pelo primeiro vídeo game que funcionava com moedas deve ir para a Computer Recreations pelo seu jogo Galaxy Game, uma outra versão reaproveitada de Spacewar! que era a única máquina em Stanford University. Ele estreou dois meses antes do Computer Space de Bushnell, mas não se aventurou mais longe do que a cafeteria do campus).

Figura 04 – Computer Space, clone do Spacewar!

Fonte: Lanzi (Acesso em: 05 agosto 2014)

Berens e Howard (2008) afirmam que o problema para Bushnell era de produzir em massa ou não: e foram fabricados 1.500 gabinetes e as vendas não eram numerosas o suficiente, algo que ele culpou tanto a falta de apoio da Nutting e a jogabilidade relativamente complicada. Destemido, ele e Dabney se separaram com Nutting em 1972 para criar a Atari (um movimento de ataque no jogo de tabuleiro japonês, Go), e nesse ano foi produzido o icônico Pong.

Lanzi (Acesso em: 05 agosto 2014) afirma que Bushnell contrata Al Alcorn para programar jogos. Desde Alcorn é inexperiente, Bushnell pediu para ele programar um simples jogo de tênis como um exercício. Eles chamam o jogo de Pong, por duas razões: "pong" é o som que o jogo faz quando a bola atinge uma raquete ou um dos lados da tela, e o nome Ping-Pong já está protegido por direitos autorais. Bushnell tenta vender Pong mas não encontra nenhum interessado, então ele decide comercializar o jogo mesmo assim. Foi feito um teste de comercialização em um bar local chamado Andy Capps. Dentro de duas semanas a unidade de teste quebra porque muitas moedas foram usadas na máquina. Pong é um sucesso.

Já Berens e Howard (2008) afirmam que problemas com a produção em massa levou a Atari a construir o seu próprio simples protótipo para um bar local, que relataram de volta dentro de duas semanas que a máquina havia quebrado. Em doze meses, a Atari vendeu em torno de 8.500 unidades de Pong, inspirando imitadores inumeráveis. Em 1973, Pong Tron, com sede no Japão. A Sega fundada na América. Soccer, da própria Taito, do Japão (outro fornecedor tradicional de máquina de arcade). Por um tempo, jogar vídeo games significava jogar Pong.

De acordo com Berens e Howard (2008), Ralph Baer, enquanto isso, convenceu seus novos empregadores que jogos baseados em TV era o futuro, e em 1967 lançou um protótipo que em 1972 foi transformado no Magnavox Odyssey, conforme a figura 05, o primeiro sistema doméstico vídeo game. Seus doze jogos, todos eram pequenas variações de um mesmo jogo - um incluído um precursor do Pong. Curiosamente, o Bushnell antes de Pong é conhecido por ter atendido a imprensa no lançamento do Odyssey e tê-lo jogado (Magnavox, que teria tido a visão para patentear a sua versão, viria a receber US $ 700.000 em danos da Atari). A confusa comercialização levou os potenciais compradores a acreditar que o sistema só

funciona em um aparelho de TV Magnavox, e a visão de Baer de vinte dólares extra teve de alguma forma tornar-se uma compra de US$ 100, mas ainda vendeu cerca de 100.000 unidades.

Figura 05 – Magnavox Odissey criado por Ralph Baer

Fonte: Lanzi (Acesso em: 05 agosto 2014)

De acordo com Berens e Howard (2008) tal como acontece com Pong, o Odyssey passou por uma situação complicada. Em 1976, os rivais incluindo Coleco's Telstar e a Fairchild Camera & Instrument's inovando com o Video Entertainment System (mais tarde renomeada para Channel F) como visto na figura 06, o primeiro console a usar cartuchos para carregar jogos em vez de utilizar fiação nos circuitos. Era um sistema primeiro copiado pela RCA (o RCA Studio II), e em seguida pela Atari em 1977, com seu VCS (*Video Computer System*, traduzindo, Sistema de Computador de Vídeo) e Atari 2600. Mas todos os jogos apresentados desenvolvidos exclusivamente para suas plataformas se saíram mal. Na verdade, o VCS foi tão mal que perdeu milhões de dólares da Warner e levou a saída de Bushnell em1978.

Figura 06 – Video Entertainment System utilizando cartuchos

Fonte: Lanzi (Acesso em: 05 agosto 2014)

Segundo Berens e Howard (2008) enquanto a década virava, as fortunas da Atari caíam dramaticamente. Mas em 1979, a Fairchild retirou-se do que eles viram como um mercado de muitos gastos, deixando Atari para vender um milhão de consoles para um público efetivamente não dividido naquele ano (Odyssey 2 da Magnavox, enquanto vendia respeitavelmente, ainda era muito pouco em comparação). Depois, em 1980, com o único concorrente sério sendo o Intellivision da Mattel, o VCS se tornou o primeiro console a receber um jogo de arcade licenciado e, portanto, o primeiro a ser capaz de reivindicar conteúdo exclusivo: Space Invaders. Só por trazer o arcade de dois anos de sucesso da Namco para a sala, adicionando várias novas opções de jogo para ele, as vendas e os lucros subiram muito, dobrando a cada ano até 1982.

De acordo com Kent (2001) Pac-Man foi a invenção de Toru Iwatani, um jovem entusiasta de pinball que se juntou a Namco pouco depois de se formar na faculdade em 1977. Iwatani queria criar máquinas de pinball, mas Namco só fabricava jogos de vídeo. Em abril de 1979, Iwatani decidiu tentar algo diferente de pinball. Ele queria fazer um jogo não violento, algo que os jogadores poderiam desfrutar. Ele decidiu construir o seu jogo em torno da palavra japonesa “taberu”, que significa "comer."

Segundo Kent (2001), Iwatani criou uma equipe de nove homens para converter o seu conceito em um jogo. A primeira coisa que ele produziu foi o personagem Pac-Man, que era um simples círculo amarelo com uma fatia cortada por uma boca. O próximo passo foi a criação de inimigos de Pac-Man. Desde que o jogo deveria apelar para o público feminino, Iwatani sentiu que os monstros tinham que ser bonitos. Então ele criou "fantasmas" coloridos que pareciam mais com esfregões com olhos grandes. O labirinto, pontos, e as “pílulas” de energia vieram em seguida, conforme a figura 07. Demorou pouco mais de um ano para produzir um protótipo funcional do jogo.

Figura 07 – O jogo Pac-Man da Namco

Fonte: Digital Games (Acesso em: 05 agosto 2014)

Kent (2001) afirma que a indústria de vídeo game mudou após o sucesso do Pac-Man. Antes de Pac-Man, o tema mais popular para jogos eram atirar em alienígenas. Depois de Pac-Man, a maioria dos jogos envolveram labirintos.

De acordo com Donovan (2010), se qualquer jogo resumiu ambos os excessos dos anos de sucesso e a dor da queda, foi E.T., The Extra-Terrestrial, o grande jogo do Atari VCS 2600 para o Natal 1982. Filme de sucesso de público no verão do mesmo ano, de Steven Spielberg, era um conto de um amigável alienígena preso na Terra, que tornou-se um dos filmes com maior bilheteria de todos os tempos. Em uma tentativa de congraçar-se com o diretor mais famoso de Hollywood, Steve Ross, presidente da Warner, fechou um acordo de 25 milhões de dólares com Spielberg para os direitos de fazer um jogo baseado no filme e, em seguida, informou a Atari do que ele tinha feito. Ray Kassar, presidente da Atari, ficou chocado dizendo que Ross teria forçado a fazer E.T. Ele o chamou e disse que tinha garantido 25 milhões de dólares a Spielberg para trabalhar neste projeto. Kassar então disse para Steve que nunca tinha garantido dinheiro para ninguém e ficou espantado com a quantia de 25 milhões dólares.

Segundo Kent (2001) E.T. se tornou famoso em toda a indústria do vídeo game pelo o seu jogo sem graça e história decepcionante. O jogo envolveu o extraterrestre de

Spielberg levando-o para vários perigos, enquanto ele tentava montar um dispositivo intergaláctico para o levar de volta para casa. Os gráficos do jogo eram primitivos, mesmo pelos padrões do Atari 2600, e E.T. passou a maior parte do jogo caindo em buracos.

De acordo com Kent (2001), cavalgando na esteira do desastre de Pac-Man (a versão ficou ruim em relação a versão original do arcade da Namco), E.T. foi demais para a Atari. A Atari tinha conseguido vender milhões de cartuchos de Pac-Man, mas a maioria dos cartuchos do E.T. permaneceram em estoque. A Atari tentou soluções para sair fora do buraco, licenciando jogos de sucesso de arcade, muitas vezes gastando milhões de dólares pelos direitos exclusivos.

Segundo Kent (2001), nem mesmo versões caseiras dos últimos sucessos de arcade ajudou. Os consumidores já tinham começado a perder o interesse em arcade, e, em 1983, eles pararam de comprar videogames. A indústria que tinha mostrado um crescimento tão milagrosa em 1982, tornou-se de repente um buraco negro.

Kent (2001) afirma que a Atari ficou com enormes estoques de cartuchos de jogos inúteis. Sem esperança de vendê-los, a Atari despejou milhões de cartuchos em um aterro sanitário no deserto do Novo México. Quando os relatórios saíram e as pessoas começaram a descobrir sobre o aterro sanitário, a Atari enviou rolos compressores para esmagar os cartuchos, então derramou cimento sobre os escombros. Até o final de 1983, a Atari tinha acumulado 536 milhões dólares em perdas. Warner Communications vendeu a empresa no ano seguinte. O mercado ficou lotado de jogos de baixa qualidade. Esse evento ficou conhecido como a grande recessão na indústria de jogos eletrônicos que ocorreu por volta de 1984.

De acordo com Donovan (2010), Donkey Kong estreou em 1981, e foi o primeiro jogo projetado por Shigeru Miyamoto, conforme a figura 08, que viria a ser considerado como um dos melhores designers de jogos do mundo. O jogo de estreia do designer japonês foi encarregado para puxar a operação da Nintendo da América para fora de um buraco.

Figura 08 – Donkey Kong, primeiro jogo de Shigeru Miyamoto

Fonte: Barton e Loguidice (2009)

De acordo com Barton e Loguidice (2009) a franquia Donkey Kong garantiu à Nintendo mais de US $ 280 milhões até 1983, e os encorajou a desenvolver outros sucessos de arcade como Mario Bros (1983), um spin-off de Donkey Kong. Também lhes deu capital para entrar no negócio de consoles domésticos, começando com o NES (*Nintendo Entertainment System*, traduzindo, Sistema de Entretenimento da Nintendo) conhecido como Famicom na Ásia, mais tarde modificado e introduzido nos Estados Unidos. O sucesso do NES e Super Mario Bros, que já vendeu mais de 40 milhões de cópias em todo o mundo, são creditados com a ressurreição o mercado de videogames moribundo depois da grande recessão na indústria de jogos eletrônicos de 1984. Super Mario Bros, conforme a figura 09, desempenhou um papel decisivo não só no renascimento do mercado de consoles, mas na expansão da indústria de vídeo games como um todo.

Figura 09 – Super Mario Bros, sucesso da Nintendo

Fonte: Barton e Loguidice (2009)

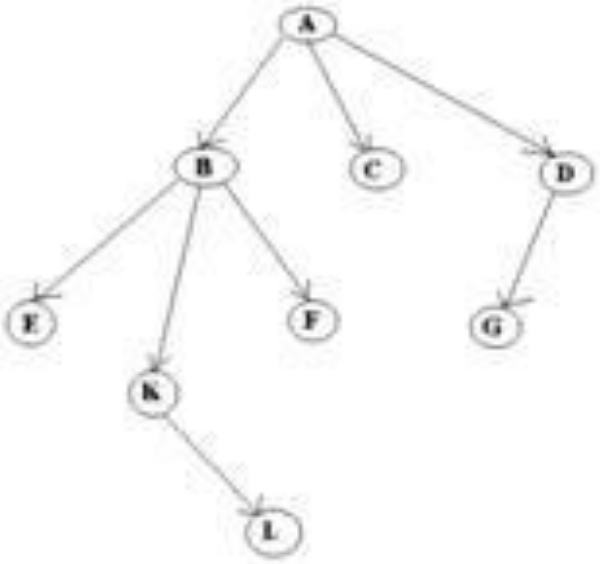
Com a forte presença da Nintendo no mercado, a indústria de vídeo games se estabilizou e conseguiu se manter desde então. Muitas outras empresas (como a Sega, Sony e Microsoft) foram entrando nesse mercado deixando-o cada vez mais competitivo. Os jogos foram evoluindo e o modo de interação com o jogador também. A evolução do hardware também contribuiu muito a transformar a indústria no que é hoje. Desde então, o mercado de jogos passou a ser um dos maiores do mundo, movimentando bilhões de dólares e a cada ano crescendo ainda mais.

2.3 Estruturas de Dados

Segundo Gunawardena (2007), há muitas estruturas de dados básicas que podem ser usadas para resolver problemas de aplicações. Matriz é uma boa estrutura de dados estáticos que podem ser acessados aleatoriamente e é bastante fácil de implementar. Listas encadeadas, por outro lado, é dinâmica e é ideal para aplicações que requerem operações frequentes, tais como inserir, excluir e atualizar. Existem outras estruturas de dados especializadas, como pilhas e filas, que permitem resolver problemas complicados, estendendo o conceito de estrutura de dados como lista encadeada, pilha e fila para uma estrutura que pode ter múltiplas relações entre seus nós. Tal estrutura é chamada de árvore. Uma árvore é um conjunto de nós conectados com arestas por direção (ou sem direção). Uma árvore é uma estrutura de dados não-linear, em comparação com matrizes, listas

encadeadas, pilhas e filas, que são estruturas de dados lineares. Uma árvore pode estar vazia com nenhum nó ou a árvore pode ser composta por um nó chamado de raiz com zero ou mais sub-árvores. A árvore tem as seguintes propriedades gerais:

Figura 10 – Estrutura de dados representando uma árvore



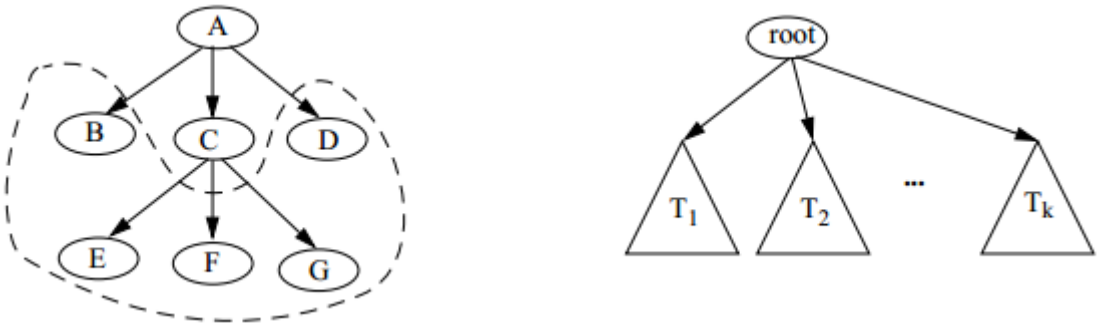
Fonte: Gunawardena (2007)

De acordo com Gunawardena (2007) como visto na figura 10, um nó é distinguido como uma raiz. Cada nó (exceto a raiz) é conectado por uma aresta direcionada exatamente de um outro nó, a direção é: pai -> filhos. A é pai de B, C, D. B é chamado de filho de A. Por outro lado, B é pai de E, F, K. Na figura acima, a raiz tem três sub-árvores. Cada nó pode ter um número arbitrário de filhos. Os nós sem filhos são chamados de folhas, ou nós externos. Os nós C, E, F, L, G são as folhas. Os nós que não são folhas, são chamados de nós internos. Os nós internos têm pelo menos um filho. Os nós com o mesmo pai são chamados de irmãos. Na figura, B, C, D são chamados de irmãos. A profundidade de um nó é o número de arestas a partir da raiz para o nó. A profundidade de K é 2. A altura de um nó é o número de arestas a partir do nó de folha mais profunda. A altura de B é 2. A altura de uma árvore é uma altura de uma raiz.

Segundo Lavender (1996) uma árvore genérica é a uma estrutura de dados que tem a nó raiz, que tem um ou mais nós filhos. Uma árvore é uma estrutura de dados recursiva, de modo que cada nó filho pode ser o nó raiz para um outro conjunto de

nós filhos, além de vários nós filhos, como mostrado na figura 11. Uma árvore com N nós tem N-1 arestas, uma vez que cada nó (exceto a raiz) tem uma aresta conectando ao nó pai.

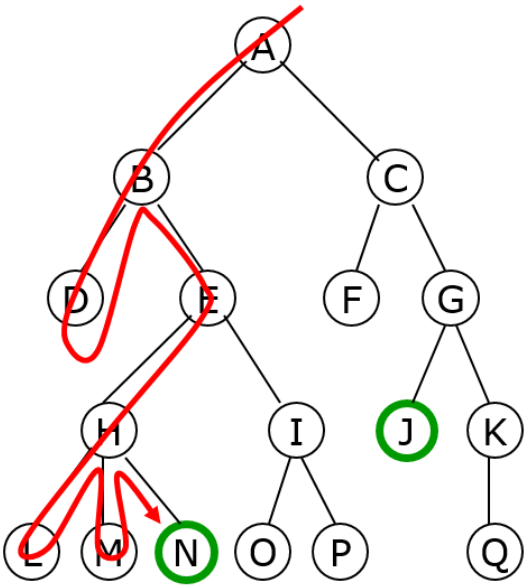
Figura 11 – Árvore genérica



Fonte: Lavender (1996)

Existem vários tipos de árvore, como por exemplo, binária, AVL, rubro-negro, B, B+, etc. Também existem vários métodos para buscar elementos e percorrer por toda a árvore, como por exemplo a busca em largura e a busca em profundidade. De acordo com Wu (2005) a busca em profundidade processará os vértices (os nós da árvore) primeiro pela profundidade e então pela largura. Depois de processar um vértice, será feito um processo recursivo com todos os seus descendentes. Como apresentado na figura 12, explora-se todo o caminho até uma folha e depois retrocede-se e explora-se o outro caminho. Os nós estão sendo explorados nesta ordem: A, B, D, E, H, L, M, N, I, O, P, C, F, G, J, K, e Q.

Figura 12 – Busca por profundidade



Fonte: University of Pennsylvania (Acesso em: 07 setembro 2014)

A utilização da estrutura de dados do tipo árvore auxiliará na resolução do problema principal descrito neste trabalho.

2.4 Considerações sobre o capítulo

O presente capítulo abordou a fundamentação teórica necessária para a compreensão dos capítulos seguintes, apresentando tópicos como a grande importância da área da Inteligência Artificial, seu crescimento ao longo dos anos e como ela tem ajudado a resolver muitos problemas. A evolução dos Jogos Digitais se deu graças à evolução do hardware. Foi apresentada ainda a estrutura de dados do tipo árvore, como é sua representação e como é feita uma busca em profundidade nela.

O capítulo seguinte aborda o algoritmo Minimax, que é um algoritmo de Inteligência Artificial para aplicar em jogos e que possui algumas características específicas, e a utilização de uma estrutura de dados do tipo árvore. Também aborda a importância da função de utilidade do algoritmo Minimax e sobre o corte alfa-beta, para otimizar o algoritmo Minimax.

3 ALGORITMO MINIMAX

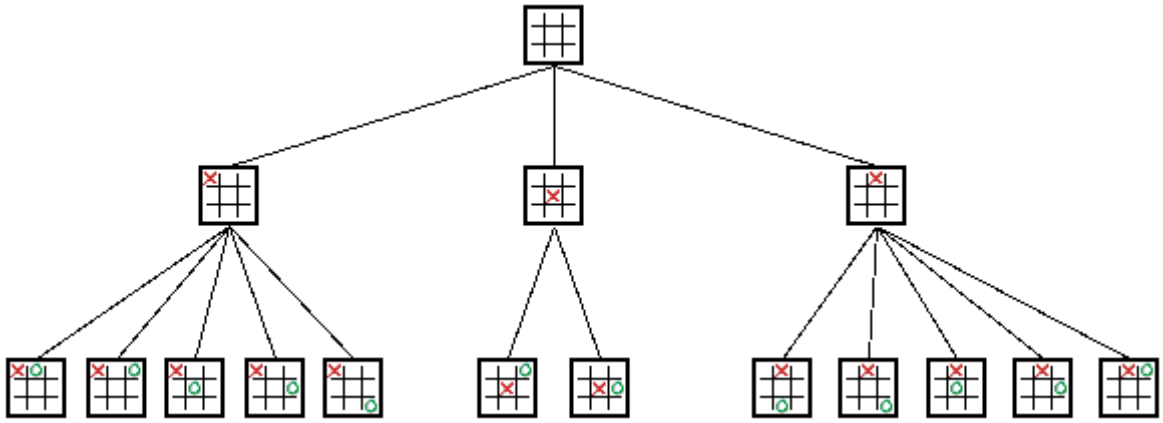
O capítulo anterior apresentou a história e importância da área da Inteligência Artificial, a evolução dos Jogos Digitais e a estrutura de dados do tipo árvore.

Neste capítulo, será apresentado o algoritmo Minimax e suas características, bem como em que tipo de jogo o algoritmo poderá ser aplicado. Serão mostrados alguns exemplos da árvore Minimax (estrutura de dados utilizada pelo algoritmo) e também o próprio algoritmo Minimax. Também aborda a importância da função de utilidade do algoritmo Minimax e sobre o corte alfa-beta, para otimizar o algoritmo Minimax.

De acordo com Pinto (2002), o algoritmo Minimax é aplicado em jogos com dois jogadores jogando por turno e revezando a vez de jogar, como o jogo da velha, damas, xadrez, go e outros. Todos estes jogos têm pelo menos uma coisa em comum: eles são jogos de lógica. Isto significa que eles podem ser descritos por um conjunto de regras. Com elas, é possível saber a partir de um determinado ponto no jogo e quais são os próximos movimentos disponíveis. Assim, estes tipos de jogos também compartilham outras características, como a informação completa (ou perfeita), ou seja, cada jogador sabe tudo sobre os movimentos possíveis do adversário. Por exemplo, em um jogo de xadrez é possível visualizar todas as peças (tanto as suas, quanto as do adversário) na mesa ou tabuleiro e fazer jogadas estratégicas com estas informações. Outra característica é do jogo ser soma-zero, ou seja, a vitória de um jogador significa a derrota de seu adversário. Apesar de terem um objetivo comum, eles nunca irão cooperar um com o outro.

De acordo com Suh (Acesso em: 27 agosto 2014), a árvore do Minimax é usada para programação de computadores para jogar jogos em que há dois jogadores se revezando para fazer suas jogadas. No sentido mais básico, a árvore Minimax é apenas uma árvore com todos os movimentos possíveis. Por exemplo, a figura 13 mostra a árvore Minimax de todos os possíveis primeiros dois movimentos do jogo da velha (a árvore foi simplificada, eliminando posições simétricas).

Figura 13 – Árvore Minimax com algumas jogadas do jogo da velha

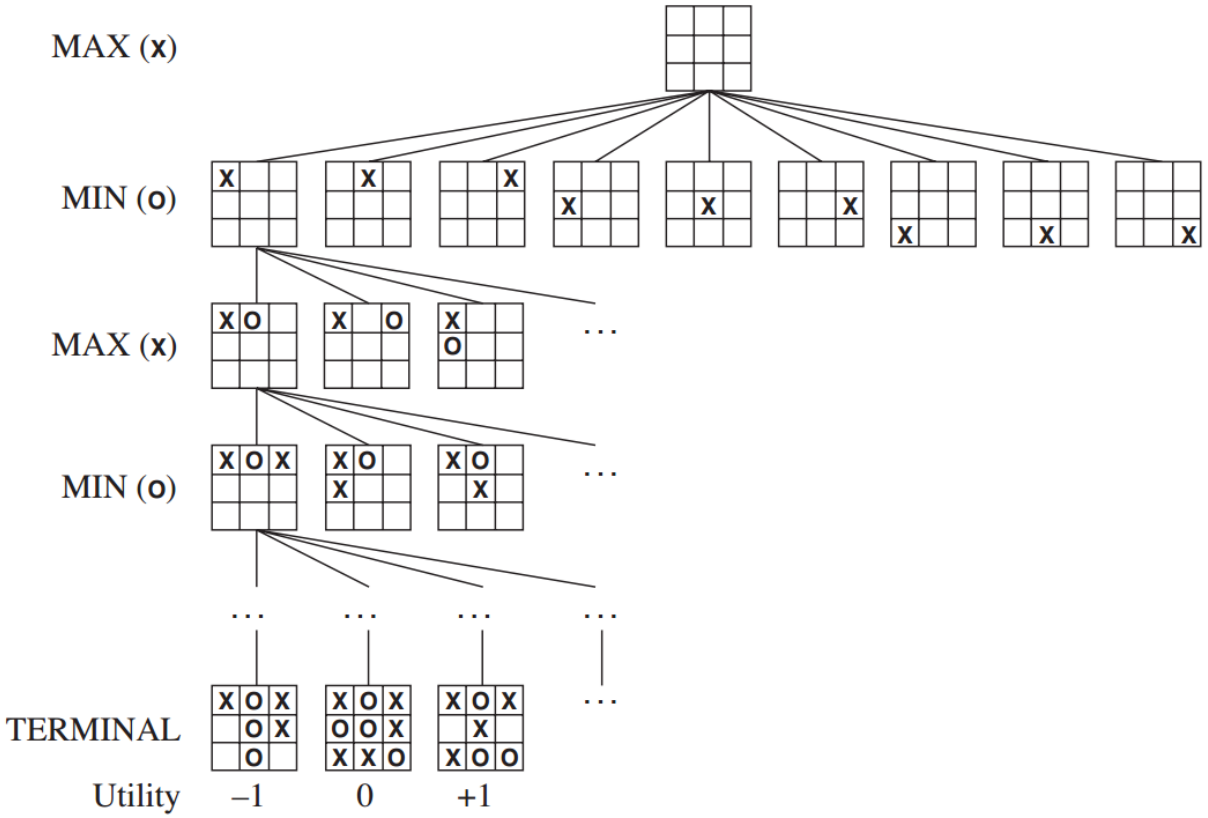


Fonte: Suh (Acesso em: 27 agosto 2014)

Segundo Suh (Acesso em: 27 agosto 2014), ao contrário de outros tipos de árvores (como árvores binárias), um nó da árvore Minimax pode ter qualquer número de filhos, dependendo da situação do jogo. Com uma árvore Minimax completa, o computador pode olhar adiante para cada movimento e determinar a melhor jogada possível. Como pode ser visto na Figura 13, a árvore pode ficar muito grande com apenas alguns movimentos, e calcular até mesmo cinco movimentos à frente pode sobrecarregar rapidamente um computador simples. Assim, para os grandes jogos como xadrez e go, os programas de computador são obrigados a estimar que está ganhando ou perdendo por amostragem, ou seja, apenas analisando uma pequena parte de toda a árvore. Além disso, existem algoritmos como Alpha-Beta pruning, Negascout e MTD (f) que contribuem na tentativa limitar o número de nós que o computador deve examinar.

De acordo com Suh (Acesso em: 27 agosto 2014), a razão pela qual esta estrutura de dados é chamada de árvore Minimax é por causa da lógica por trás da estrutura. Atribuindo pontos para o resultado de um jogo da velha: se X ganha, a situação do jogo é dada o valor de 1. Se O vence, o jogo tem um valor de -1. Agora, X vai tentar maximizar o valor do ponto, enquanto que O vai tentar minimizar o valor do ponto. Assim, um dos primeiros pesquisadores na árvore Minimax decidiu o nome do jogador X como Max e jogador O como Min. Assim, a estrutura de dados inteira passou a ser chamada de árvore Minimax.

Figura 14 – Árvore Minimax com MAX/MIN de uma partida parcial do jogo da velha



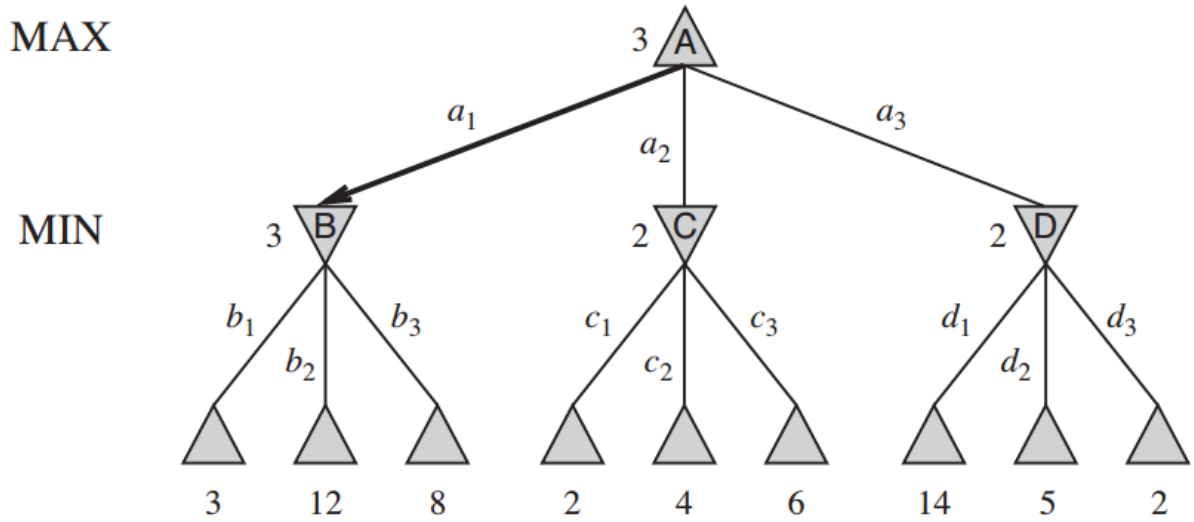
Fonte: Norvig e Russell (2009)

Segundo Norvig e Russell (2009), a figura 14 mostra uma árvore (parcial) de uma partida do jogo da velha. O nó superior é o estado inicial, e o jogador MAX faz sua jogada em primeiro lugar, colocando um X em um quadrado vazio. Está sendo mostrada parte da árvore, e as jogadas estão sendo alternadas pelo jogador MIN (no caso, O) e o jogador MAX (no caso, X), até que, eventualmente, alcance os estados terminais (que são estados em que a partida é finalizada), que podem ser atribuídos valores de utilidades, definidos de acordo com as regras do jogo. No exemplo, -1 o jogador O vence, 0 o jogo empata e +1 o jogador X vence.

De acordo com Norvig e Russell (2009), em um problema de busca normal, a solução ideal seria uma sequência de ações que levam a um estado objetivo, ou seja, o estado terminal que é uma vitória. MAX deve encontrar uma estratégia contingente, que especifica a jogada de MAX no estado inicial e então as jogadas de MAX nos estados resultantes de cada resposta possível por MIN. Em seguida, são computadas as jogadas de MAX nos estados resultantes de cada resposta possível pelo MIN para esses movimentos, e assim por diante. Mesmo para um jogo simples

como o jogo da velha, é muito complexo mostrar toda a árvore do jogo. Por isso, a figura 15 apresenta a evolução da árvore em um jogo trivial qualquer. Os movimentos possíveis para MAX no nó raiz são rotulados como a1, a2 e a3. As possíveis respostas ao a1 para MIN são b1, b2, b3, e assim por diante. Este jogo particular termina após cada jogada feita por MAX e MIN. Os valores de utilidades dos estados terminais neste jogo variam de 2 a 14. A figura 15 mostra que os nós ▲ (triângulo apontando para cima) são os nós do MAX, na qual é a vez do MAX fazer sua jogada, e os nós ▼ (triângulo apontando para baixo) são os nós do MIN. Os nós terminais mostram os valores de utilidade para MAX. Os outros nós são rotulados com seus valores Minimax. A melhor jogada do MAX na raiz é a1, porque leva ao estado com o maior valor Minimax, e a melhor resposta do MIN é b1, porque leva ao estado com o menor valor Minimax.

Figura 15 – Árvore Minimax de um jogo trivial

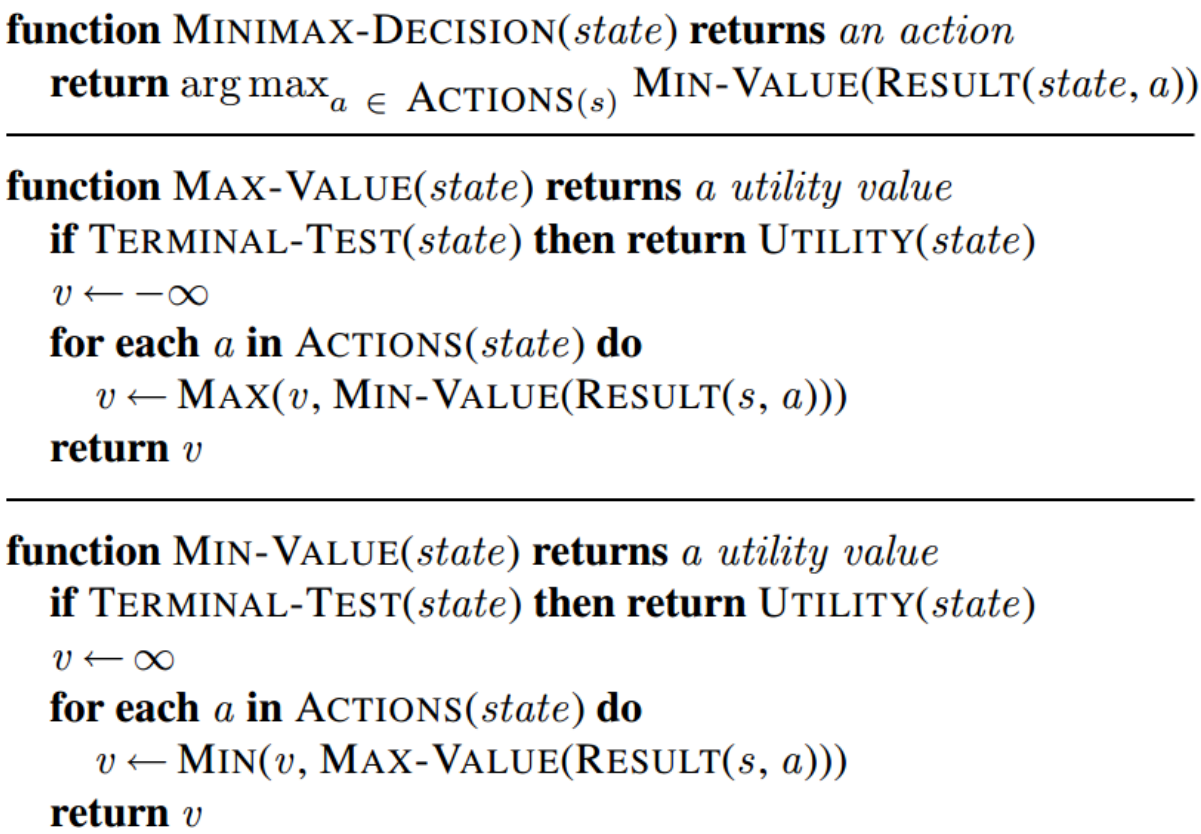


Fonte: Norvig e Russell (2009)

O algoritmo Minimax, como mostrado na figura 15, calcula a decisão Minimax do estado atual. Ele usa um simples cálculo recursivo dos valores Minimax de cada estado sucessor, diretamente implementando as equações que definem. A recursão prossegue por todo o caminho até as folhas da árvore. Em seguida, os valores Minimax são copiados através da árvore enquanto a recursão se desenrola. Por exemplo, o algoritmo faz a recursividade para os três nós do canto inferior esquerdo e usa a função de utilidade com eles para descobrir que os seus valores são de 3, 12 e 8, respectivamente. Em seguida, é descoberto que o menor valor entre eles é

3, e retorna este valor para o nó B. O processo é semelhante para calcular os valores de 2 para C e de 2 para D. Por fim, é obtido o maior valor entre 3, 2 e 2 (no caso, 3) e passa-se este valor para o nó raiz A. O algoritmo Minimax realiza uma completa busca em profundidade na árvore do jogo. Se a profundidade máxima da árvore é *a* e há *b* jogadas legais em cada ponto, então a complexidade de tempo do algoritmo Minimax é $O(b^a)$. A complexidade de espaço é *O(ba)* para um algoritmo que gera todas as jogadas de uma só vez, ou *O(a)* para um algoritmo que gera as jogadas, uma de cada vez. Para os jogos reais, é claro, o tempo e o custo é totalmente impraticável, mas este algoritmo serve como a base para a análise matemática de jogos e para algoritmos mais práticos.

Figura 16 – Algoritmo Minimax



Fonte: Norvig e Russell (2009)

Com o algoritmo Minimax, conforme visto na figura 16, aplicado em um jogo, será possível resolver o problema de decisão de jogada.

3.1 A função de utilidade

De acordo com El-Basuny (Acesso em: 19 outubro 2014), a função de utilidade (ou também chamada de função de avaliação) tem como principal objetivo avaliar o quanto uma jogada é boa para um determinado jogador e desta forma calcular um valor heurístico para aquela jogada. Valores positivos significam vantagens para o jogador MAX, valores negativos significam vantagens para o jogador MIN.

O algoritmo Minimax é um algoritmo genérico para resolver estes problemas, mas a função de utilidade, especificamente, deve ser implementada de acordo com o jogo, ou seja, cada jogo a função deve ser totalmente diferente.

Por exemplo, em jogos mais simples como o jogo da velha, existem 3 tipos de jogadas. O jogador X venceu, considerando que seja o jogador MAX, então a função retorna um valor positivo, como por exemplo 1. O jogador O venceu, considerando que seja o jogador MIN, então a função retorna um valor negativo, como por exemplo -1. E caso o jogo empate ou ainda não chegou em algum estado terminal do jogo, retorna um valor neutro, como por exemplo 0.

O algoritmo Minimax é aplicado em jogos que possui a característica de soma zero, ou seja, em jogos competitivos: se um jogador ganha, automaticamente o outro jogador perde. Com isso é possível utilizar a mesma função de avaliação para ambos jogadores, apenas retornando um valor positivo para MAX e negativo para MIN.

No jogo Triple Triad (mais informações do funcionamento e das regras do jogo no capítulo 4) a função de utilidade foi implementada da seguinte forma: defensivo e ao mesmo tempo ofensivo. Defensivo pois o jogador artificial posiciona suas cartas de forma que os lados com valores mais fracos ficam totalmente protegidos e só expondo os valores mais fortes para o jogador adversário (heurística calculada com base nestes valores fortes). Ofensivo pois o jogador artificial terá maior interesse em capturar cartas mais fortes do que cartas mais fraca (heurística calculada com base na soma dos valores de força das cartas). E existem jogadas em que o jogador

artificial ao mesmo tempo captura uma carta e ainda protege os lados mais fracos, deixando ainda mais difícil para o jogador adversário conseguir se recuperar.

3.2 O corte alfa-beta

De acordo com Rosen (Acesso em: 13 outubro 2014), o algoritmo Minimax é uma maneira de encontrar uma jogada ideal em um jogo de dois jogadores. O corte alfa-beta é uma maneira de encontrar a solução Minimax ideal, evitando procurar em sub-árvores de jogadas que não serão mais consideradas.

Segundo Anene, Mayefsky e Sirota (Acesso em: 13 outubro 2014), o corte alfa-beta é uma melhoria sobre o algoritmo Minimax. O problema com Minimax é que o número de estados do jogo que devem ser analisados é exponencial em relação ao número de movimentos. Embora seja impossível eliminar completamente o expoente, somos capazes de cortá-lo ao meio. É possível calcular a decisão Minimax correta, sem analisar cada nó na árvore. Eliminando as possibilidades que podem ser consideradas sem ter que examiná-las, o algoritmo permite descartar grandes partes da árvore. Quando aplicado à uma árvore Minimax padrão, é calculado o mesmo como no Minimax, mas são removidas as ramificações da árvore que não pode influenciar a decisão final. O algoritmo corte alfa-beta pode ser aplicada à árvore de qualquer profundidade e que muitas vezes permite cortar sub-árvores inteiras em vez de apenas folhas. Aqui está o algoritmo geral:

1. Considere um nó *n* em algum lugar na árvore, que possa ser analisado;
2. Se houver uma melhor escolha *m*, no pai do nó *n* ou em qualquer nó até este ponto, *n* nunca será analisada;
3. Uma vez que já se tem as informações suficientes sobre *n* para chegar nesta conclusão, será cortada;

Sendo que: α (alfa) é o valor da melhor escolha que foi encontrada até agora em qualquer ponto ao longo do caminho de escolha para MAX e β (beta) é o valor da melhor escolha que foi encontrada até agora em qualquer ponto de escolha ao longo do caminho para MIN.

De acordo com Lin (2003), o algoritmo Minimax com o corte alfa-beta:

```
alpha-beta(player,board,alpha,beta)
    if(game over in current board position)
        return winner

    children = all legal moves for player from this board
    if(max's turn)
        for each child
            score = alpha-beta(other player,child,alpha,beta)
            if score > alpha then
                alpha = score (we have found a better best move)
            if alpha >= beta then
                return alpha (cut off)
        return alpha (this is our best move)
    else (min's turn)
        for each child
            score = alpha-beta(other player,child,alpha,beta)
            if score < beta then
                beta = score (opponent has found a better worse move)
            if alpha >= beta then
                return beta (cut off)
        return beta (this is the opponent's best move)
```

É necessário manter o controle de dois números (alfa e beta) para cada nó que será analisado. Alfa terá o valor da melhor jogada possível de MAX, que foi calculado até o momento. Beta terá o valor da melhor jogada possível de MIN, que foi calculado até o momento. Se a qualquer momento, alpha for maior ou igual ao beta, então a melhor jogada do jogador MAX pode forçar uma situação pior do que melhor jogada do jogador MIN até agora, e por isso não há necessidade de avaliar mais esta jogada. A mesma condição é aplicada se for o jogador MIN, exceto em vez de encontrar a jogada que produz alfa, irá encontrar a jogada que produz beta. Para garantir que este algoritmo retorne um valor, alfa começa com valor igual a ∞ negativo e beta com valor igual a ∞ positivo, e atualizar esses valores ao analisar mais nós.

Com o corte alfa-beta aplicado no algoritmo Minimax, a busca por soluções ótimas serão muito mais rápidas.

3.3 Considerações sobre o capítulo

O presente capítulo abordou o algoritmo Minimax, descrevendo como é a árvore Minimax e mostrando, por exemplo, algumas possíveis jogadas de um jogo da velha

com jogadores representados por MAX e MIN, que devem maximizar e minimizar, respectivamente, o valor de utilidade que representará para qual jogada o jogador da decisão terá que seguir, com base na estrutura da árvore do Minimax. Foi apresentado também o próprio algoritmo Minimax, que é recursivo, como também a complexidade de execução e de espaço que pode afetar o desempenho do algoritmo, dependendo da quantidade de jogadas que será calculada. Também abordou a importância da função de utilidade do algoritmo Minimax e sobre o corte alfa-beta, para otimizar o algoritmo Minimax.

O capítulo seguinte aborda o jogo que será desenvolvido no presente trabalho, e como o algoritmo Minimax será aplicado para resolver o problema principal do trabalho.

4 ESTUDO DE CASO

O capítulo anterior apresentou as informações do algoritmo Minimax, que tem grande importância no presente capítulo. Também apresentou a importância da função de utilidade do algoritmo Minimax e sobre o corte alfa-beta, para otimizar o algoritmo Minimax.

Neste capítulo, será apresentado o jogo Triple Triad, incluindo sua origem, regras e funcionamento do jogo. Também será descrito e comentado o código fonte com algoritmo Minimax aplicado ao Triple Triad, criando um oponente artificial para um jogador humano, e o código fonte do jogo da velha com o Minimax aplicado e comentado.

4.1 O jogo Triple Triad

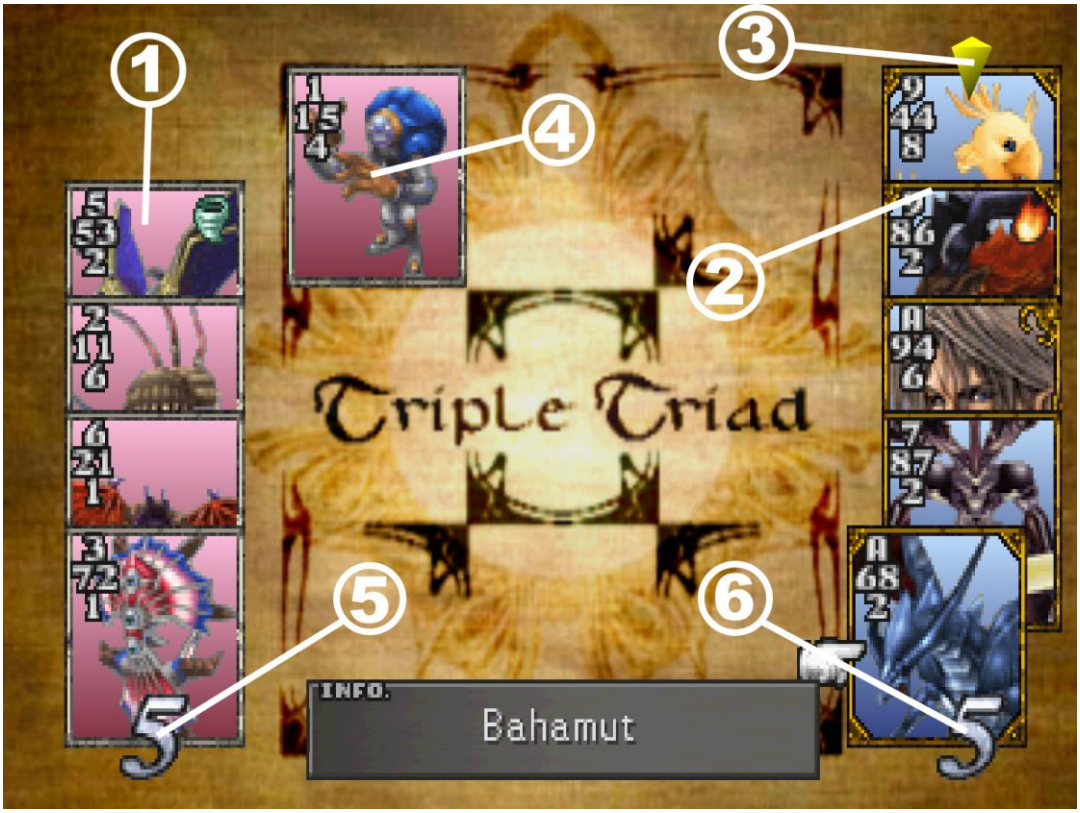
O Triple Triad é um minijogo de cartas que faz parte de outro jogo, ou seja, não é um jogo *standalone* (significa que não depende de outra aplicação para seu funcionamento, é autossuficiente), chamado Final Fantasy VIII, Squaresoft (1999), um jogo do tipo RPG (*Role-Playing Game*, traduzindo, Jogo de Interpretação de Personagem), originalmente desenvolvido e publicado em 1999 para o console Sony Playstation, pela Squaresoft (hoje Square Enix), empresa japonesa que desenvolve vários jogos como a franquia Final Fantasy e muitas outras conhecidas no mercado de jogos por todo o mundo.

Triplo Triad é jogado em uma grade quadrada de três por três (3x3) de espaços em branco, onde as cartas serão colocadas conforme o jogo progride. As cartas retratam vários personagens, monstros e chefes de Final Fantasy VIII. Cada carta tem quatro números colocados no canto superior esquerdo e que representam a força da carta. Cada número corresponde a um dos quatro lados da carta. Estes números variam de 1 a 9 e a letra A representa o número 10. Há duas cores de carta – vermelho e azul – que determinam a qual jogador a carta pertence naquele momento. Em um jogo básico de Triplo Triad, cada jogador tem cinco cartas. É feita uma decisão aleatória para decidir qual dos dois jogadores começará. O jogador que ganhar o sorteio pode, então, escolher uma carta para jogar em qualquer lugar na

mesa. Após a primeira carta jogada, o jogador adversário pode então jogar uma carta em qualquer espaço desocupado no tabuleiro. O jogo continua com turnos dos jogadores alternando desta forma. Não existe nenhuma restrição relacionada ao tempo no jogo, ou seja, uma ação poderá tomar o tempo necessário para ser realizada. As imagens deste capítulo foram obtidas diretamente do próprio jogo original.

Uma partida comum no Triple Triad consiste, de acordo com a Figura 17:

Figura 17 – Partida do jogo Triple Triad



Fonte: Elaborado pelo próprio autor

1. Cartas do jogador vermelho;
2. Cartas do jogador azul;
3. Indicador amarelo, indicando o turno do jogador que irá fazer a jogada;
4. Carta jogada na mesa e que no momento pertence ao jogador vermelho;
5. Quantidade de cartas capturas pelo jogador vermelho;
6. Quantidade de cartas capturas pelo jogador azul.

As cartas são representadas da seguinte maneira, de acordo com a Figura 18:

Figura 18 – Informações de uma carta

Fonte: Elaborado pelo próprio autor

1. Representa a força da carta, que pode variar de 1 (mais fraco) à 9, e A sendo o valor mais forte. Existem 4 valores, pois representam a força para aquele lado. No exemplo acima, A no superior, 6 na esquerda, 8 na direita e 2 no inferior;
2. Indica a cor do jogador a quem a carta pertence: vermelho ou azul. Desta forma, a carta apresentada na figura pertence ao jogador azul;
3. Representa a imagem do personagem ou criatura, sendo que estes fazem parte do mundo de Final Fantasy VIII.

Cada jogador tem seu turno para jogar uma carta até que os 9 espaços da mesa 3x3 sejam preenchidos e o jogo se encerre. O primeiro jogador é escolhido de forma aleatória no início da partida.

Para ganhar o jogo, a maioria (do total de dez) das cartas jogadas (inclusive a última carta que não é colocada na mesa) deve ser da cor do jogador correspondente (no caso, vermelho ou azul). Para fazer isso, o jogador deve capturar cartas. Para

capturar uma carta, o jogador deve colocá-las adjacentes às cartas do oponente e então os valores de força dos lados em que as duas cartas que entraram em contato serão comparadas. Se o valor de força da carta do jogador é maior do que a carta adjacente, ela será capturada e modificada para a cor do jogador que a capturou. A captura só pode ocorrer durante o turno daquele jogador, e o adversário não poderá capturar uma carta durante este turno. O empate ocorrerá se os jogadores possuírem o mesmo número de cartas na sua cor no tabuleiro, ao terminar a partida.

O processo de captura de uma carta é feito da seguinte maneira, de acordo com a Figura 19:

Figura 19 – Capturando uma carta

Fonte: Elaborado pelo próprio autor

1. O jogador vermelho colocou a carta na mesa;
2. Agora é o turno do jogador azul e será colocada uma carta no espaço indicado;
3. No momento que é colocado a carta do jogador azul, são verificadas todas as cartas vizinhas para detectar alguma carta do oponente vermelho. No caso acima, existe uma carta na posição superior (carta do jogador vermelho) e, neste caso é verificado o valor de força do lado do conflito entre as cartas. O valor A (que vale 10) no superior da carta azul é maior que o 4 inferior da carta vermelha do oponente, conforme indicado pelas setas. Uma observação importante é que se no lugar do A superior da carta azul fosse 4 ou menos, o

jogador azul não perderia sua carta para o oponente vermelho, ou seja, na vez do jogador não é possível perder suas próprias cartas capturadas. O que poderá acontecer é empatar (mantendo o jogo do mesmo jeito) ou capturar cartas do oponente. É possível em uma jogada capturar várias cartas do oponente;

4. Carta foi capturada com sucesso e neste momento a carta vermelha passou ser azul, aumentando, assim, o contador de cartas capturadas do jogador azul.

Desta forma é definida a regra natural e implícita do jogo, pelo processo de captura em que o maior valor da carta vence.

A regra Open tem como objetivo os jogadores mostrarem todas as cartas em jogo para o outro. Desta forma o jogador poderá prever jogadas e estratégias do oponente. Essa regra só afeta o jogador humano, pois o jogador artificial já sabe quais são suas cartas de qualquer jeito.

Como pode ser visto na figura 20, as cartas do jogador à esquerda (jogador artificial) da tela estão todas viradas e só serão mostradas quando forem jogadas na mesa:

Figura 20 – Regra Open

Fonte: Elaborado pelo próprio autor

Existe um total de 110 cartas divididas por níveis de 1 a 10 com cada nível tendo 11 cartas. Cartas de nível 1 possuem valores de força como 1, 2 ou 3, enquanto cartas de nível 10 tem valores de força como 8, 9 ou A.

As imagens nas cartas representam entidades no mundo de Final Fantasy VIII, como personagens e criaturas. As cartas de nível 1, 2, 3, 4 e 5 representam monstros fracos. As cartas de nível 6 e 7 representam chefes ou monstros fortes e únicos. As cartas de nível 8 e 9 representam criaturas extremamente poderosas que são invocadas pelos personagens, e normalmente possuem 2 lados fortes e 2 fracos. As cartas de nível 10 representam os personagens e normalmente possuem 3 lados fortes e 1 fraco. As bordas e contornos são diferentes, de acordo os níveis das cartas, como mostra a figura 21, que apresenta cartas da esquerda para a direita, de níveis 1, 6, 9, e 10:

Figura 21 – Cartas de diferentes níveis

Fonte: Elaborado pelo próprio autor

4.2 Desenvolvimento do jogo Triple Triad

Nesta seção do trabalho, será descrito e comentado o código fonte do jogo Triple Triad, em que o algoritmo Minimax foi aplicado. O jogo poderá ser baixado no link: http://goo.gl/S19ihi
Será necessário baixar o instalador da DirectX da Microsoft, no seguinte link: http://goo.gl/s5Snj

```
001 int Minimax::Value(GameState& game_state,const bool
maximize_player,const int depth)
002 {
003     const Player* player_winner = game_state.CheckForWinnerCondition();
004
```

```
005     if ((depth == 0) || (player_winner != nullptr) ||
(game_state.IsBoardFull()))
006     {
007         const int winner_bonus_score = 1000;
008
009         int score = game_state.ScoreBetweenPlayerMaxAndPlayerMin() *
winner_bonus_score;
010
011         if (game_state.IsPlayerMaxWinner(player_winner))
012         {
013             return score;
014         }
015         else if (game_state.IsPlayerMinWinner(player_winner))
016         {
017             return score;
018         }
019
020         if (maximize_player)
021         {
022             return game_state.CheckForGoodMoves(game_state.player_max);
023         }
024         else
025         {
026             return -
game_state.CheckForGoodMoves(game_state.player_min);
027         }
028     }
029
030     std::vector<Card*>& current_player_cards = maximize_player ?
game_state.player_max_cards : game_state.player_min_cards;
031
032     if (maximize_player)
033     {
034         int score = INT_MIN;
035         for (auto& player_card_iterator : current_player_cards)
036         {
037             if (!player_card_iterator->played_on_board)
038             {
039                 for (size_t i = 0; i < game_state.board_cards.size();
i++)
040                 {
041                     auto& board_card_iterator =
game_state.board_cards[i];
042
043                     if (!board_card_iterator)
044                     {
045                         board_card_iterator = player_card_iterator;
046                         board_card_iterator->played_on_board = true;
047
game_state.CheckForBasicRule(*board_card_iterator,i);
048
049                         int temp_score =
Minimax::Value(game_state,false,depth - 1);
050
051
game_state.RestoreCapturedCards(*board_card_iterator);
052                         board_card_iterator->played_on_board = false;
053                         board_card_iterator = nullptr;
054
055                         if (temp_score > score)
056                         {
```

```
057                                 score = temp_score;
058                             }
059                         }
060                     }
061                 }
062             }
063             return score;
064         }
065         else
066         {
067             int score = INT_MAX;
068             for (auto& player_card_iterator : current_player_cards)
069             {
070                 if (!player_card_iterator->played_on_board)
071                 {
072                     for (size_t i = 0; i < game_state.board_cards.size();
i++)
073                     {
074                         auto& board_card_iterator =
game_state.board_cards[i];
075
076                         if (!board_card_iterator)
077                         {
078                             board_card_iterator = player_card_iterator;
079                             board_card_iterator->played_on_board = true;
080
game_state.CheckForBasicRule(*board_card_iterator,i);
081
082                             int temp_score =
Minimax::Value(game_state,true,depth - 1);
083
084
game_state.RestoreCapturedCards(*board_card_iterator);
085                             board_card_iterator->played_on_board = false;
086                             board_card_iterator = nullptr;
087
088                             if (temp_score < score)
089                             {
090                                 score = temp_score;
091                             }
092                         }
093                     }
094                 }
095             }
096             return score;
097         }
098 }
099
100 void Minimax::ComputerMove(TT::Card* current_card,vector<TT::Card*>&
current_game_cards,const Board& board,Player& player_min,Player&
player_max,const Card*& best_card,int& best_position,int& score,const int
depth)
101 {
102     game_state.player_min = &player_min;
103     game_state.player_max = &player_max;
104     game_state.CopyCards(current_game_cards);
105     game_state.SetPlayerCards();
106     game_state.SetBoardCards(board);
107
108     Card* converted_current_card =
game_state.ConvertCurrentCard(current_card);
```

```
109
110     auto& player_card_iterator = converted_current_card;
111
112     if (!player_card_iterator->played_on_board)
113     {
114         for (size_t i = 0; i < game_state.board_cards.size(); i++)
115         {
116             auto& board_card_iterator = game_state.board_cards[i];
117
118             if (!board_card_iterator)
119             {
120                 board_card_iterator = player_card_iterator;
121                 board_card_iterator->played_on_board = true;
122                 game_state.CheckForBasicRule(*board_card_iterator,i);
123
124                 int temp_score =
Minimax::Value(game_state,false,depth);
125
126                 game_state.RestoreCapturedCards(*board_card_iterator);
127                 board_card_iterator->played_on_board = false;
128                 board_card_iterator = nullptr;
129
130                 if (temp_score > score)
131                 {
132                     score = temp_score;
133
134                     best_position = i;
135                     best_card = player_card_iterator;
136                 }
137             }
138         }
139     }
140 }
```

O algoritmo tentará fazer combinações com todas as jogadas possíveis (até uma certa profundidade, devido ao custo computacional envolvido no processo), e começa chamando a função ComputerMove na linha 100. Entre as linhas 102 e 106, está sendo guardado as referências dos jogadores MIN e MAX, copiando as informações das cartas para a estrutura que será utilizada pelo algoritmo, atribuindo as cartas copiadas para os jogadores e depois para a mesa. Entre as linhas 108 e 110, está sendo feito o processo de conversão da carta que será testada, da estrutura do jogo para a estrutura especifica utilizada pelo algoritmo. Na linha 112 verificará se a carta já não está na mesa. Na linha 114 começa o loop por todas as cartas da mesa. Na linha 116 obtém a referência da carta da mesa no loop e na linha 118 verificará se possui carta ou não, nesta posição da mesa. Na linha 120 o espaço da mesa receberá a carta que será analisa (carta que está na mão do jogador). Na linha 121 será marcado que a carta da mão foi jogada na mesa. Na linha 122 será verificado pelas regras do jogo e será associado à carta verificada,

todas as cartas que ela capturará. Na linha 124 será chamado o método Value, passando o objeto do estado do jogo, um indicador se vai verificar pelo jogador MIN ou MAX e a profundidade da árvore (até aonde o algoritmo irá), retornando uma pontuação. Entre as linhas 126 e 128 está sendo feito um reset nos valores da carta da mesa sendo analisada. Entre as linhas 130 e 136 será verificado se a pontuação é maior que a melhor pontuação atual: se for, então atualizar a própria pontuação, a melhor posição com a posição da mesa atual, e a melhor carta com a carta da mão que está sendo verificada neste momento.

Na linha 124 o método Value será chamado e na linha 03 será armazena a referência do jogador que venceu, se existir algum, senão será nulo. Na linha 09 será calculado a pontuação com base na diferença de cartas viradas entre o jogador MAX e MIN. Entre as linhas 11 e 18, será verificado se o jogador MAX ou jogador MIN venceu, retornando a pontuação previamente calculada. Entre as linhas 20 e 26 será chamado o método CheckForGoodMoves para analisar o quanto vale o estado do jogo neste momento e se for o jogador MAX, retornará um valor positivo, se for o jogador MIN, retornará um valor negativo. O método CheckForGoodMoves é extremamente importante pois é aí que será definido o quanto tal jogada valerá apena ou não para o jogador MAX. Na linha 30 será obtido a referência da lista de cartas da mão do jogador atual. Na linha 32 será verificado se vai utilizar jogadas com o jogador MAX ou MIN.

Na linha 34 está sendo declarado uma variável para armazenar a maior pontuação das jogadas em diante. Na linha 35 será feito um loop pelas cartas do jogador atual. Na linha 37 será verificado se tal carta das cartas do jogador atual já não está na mesa. Na linha 39 será feito outro loop pelas cartas da mesa. Na linha 41 obtém a referência da carta da mesa no loop e na linha 43 verificará se possui carta ou não, nesta posição da mesa. Na linha 45 o espaço da mesa receberá a carta que será analisa (carta do jogador atual). Na linha 46 será marcado que a carta da mão do jogador atual foi jogada na mesa. Na linha 47 será verificado pelas regras do jogo e será associado à carta verificada, todas as cartas que ela capturará. Na linha 49 será chamado o método Value, passando o objeto do estado do jogo, um indicador que vai verificar pelo jogador MIN e a profundidade da árvore (até aonde o algoritmo irá), retornando uma pontuação. Entre as linhas 51 e 53 está sendo feito um reset

nos valores da carta da mesa sendo analisada. Entre as linhas 55 e 58 será verificado se a pontuação é maior que a pontuação atual: se for, então atualizar a própria pontuação. Na linha 63 é retornado esta pontuação.

Na linha 67 está sendo declarado uma variável para armazenar a menor pontuação das jogadas em diante. Na linha 68 será feito um loop pelas cartas do jogador atual. Na linha 70 será verificado se tal carta das cartas do jogador atual já não está na mesa. Na linha 72 será feito outro loop pelas cartas da mesa. Na linha 74 obtém a referência da carta da mesa no loop e na linha 76 verificará se possui carta ou não, nesta posição da mesa. Na linha 78 o espaço da mesa receberá a carta que será analisa (carta do jogador atual). Na linha 79 será marcado que a carta da mão do jogador atual foi jogada na mesa. Na linha 80 será verificado pelas regras do jogo e será associado à carta verificada, todas as cartas que ela capturará. Na linha 82 será chamado o método Value, passando o objeto do estado do jogo, um indicador que vai verificar pelo jogador MAX e a profundidade da árvore (até aonde o algoritmo irá), retornando uma pontuação. Entre as linhas 84 e 86 está sendo feito um reset nos valores da carta da mesa sendo analisada. Entre as linhas 88 e 91 será verificado se a pontuação é menor que a pontuação atual: se for, então atualizar a própria pontuação. Na linha 96 é retornado esta pontuação.

No final do processo, a variável de melhor posição terá o valor da posição da mesa com a melhor jogada, junto com a variável de melhor carta, e com isso o jogador artificial, fará sua jogada nesta posição, utilizando esta carta.

4.3 Desenvolvimento do jogo da velha

Nesta seção do trabalho, será descrito e comentado o código fonte do jogo da velha, em que o algoritmo Minimax foi aplicado e auxiliou na conclusão deste trabalho:

```
01  int MinimaxValue(Game& game,const bool maximize_player,const int depth)
02  {
03      const Player& current_player = maximize_player ?
game.player_computer : game.player_human;
04
05      const Player* player_winner = game.CheckForWinnerCondition();
06
07      if ((depth == 0) || (player_winner != nullptr))
```

```
08      {
09          if (game.IsPlayerComputer(player_winner))
10          {
11              return static_cast<int>(Minimax::GoodForPlayerComputer);
12          }
13          else if (game.IsPlayerHuman(player_winner))
14          {
15              return static_cast<int>(Minimax::BadForPlayerComputer);
16          }
17          return static_cast<int>(Minimax::MatchDraw);
18      }
19
20      if (maximize_player)
21      {
22          int score = INT_MIN;
23          for (unsigned int i = 0; i < game.board.symbols.size(); i++)
24          {
25              if (game.board.symbols[i] == Symbol::Empty)
26              {
27                  game.board.symbols[i] = current_player.symbol;
28                  int temp_score = MinimaxValue(game,false,depth - 1);
29                  game.board.symbols[i] = Symbol::Empty;
30                  if (temp_score > score)
31                  {
32                      score = temp_score;
33                  }
34              }
35          }
36          return score;
37      }
38      else
39      {
40          int score = INT_MAX;
41          for (unsigned int i = 0; i < game.board.symbols.size(); i++)
42          {
43              if (game.board.symbols[i] == Symbol::Empty)
44              {
45                  game.board.symbols[i] = current_player.symbol;
46                  int temp_score = MinimaxValue(game,true,depth - 1);
47                  game.board.symbols[i] = Symbol::Empty;
48                  if (temp_score < score)
49                  {
50                      score = temp_score;
51                  }
52              }
53          }
54          return score;
55      }
56  }
57
58  void ComputerMove(Game& game)
59  {
60      int position = INT_MIN;
61      int score = INT_MIN;
62      for (unsigned int i = 0; i < game.board.symbols.size(); ++i)
63      {
64          if (game.board.symbols[i] == Symbol::Empty)
65          {
66              game.board.symbols[i] = Symbol::Circle;
67              int temp_score = MinimaxValue(game,false,-1);
68              if (temp_score > score)
```

```
69                 {
70                     score = temp_score;
71                     position = i;
72                 }
73                 game.board.symbols[i] = Symbol::Empty;
74             }
75         }
76         game.board.symbols[position] = Symbol::Circle;
77     }
```

O algoritmo tentará fazer combinações com todas as jogadas possíveis, e começa chamando a função ComputerMove na linha 58. É passado como parâmetro o objeto game que contém informações como jogadores, o estado da mesa marcados com X e O, etc. Nas linhas 60 e 61, são declaradas variáveis para armazenar a posição com a melhor jogada e a maior pontuação que o algoritmo Minimax irá retornar. Na linha 62, inicia o loop para cada espaço na mesa (no caso do jogo da velha, 9). Na linha 64, se a posição da mesa estiver vazia. Na linha 66, a posição vazia recebe o símbolo O (do computador, que é o jogador MAX). Na linha 67, será chamado a função MinimaxValue, passando o objeto do jogo, um campo indicando que será utilizado o jogador MIN, a profundidade da árvore será ignorada (-1) e assim retornando uma pontuação. Entre as linhas 68 e 72 é verificado se a pontuação retornada é maior que a pontuação atual armazenada: se for, então atualiza-la e também atualizar a posição da mesa, indicando que é uma boa jogada. Na linha 73, a posição novamente é resetada com um símbolo vazio, permitindo no loop seguinte tentar novas possibilidades de jogadas. E na linha 76, o jogador artificial O (MAX) faz sua jogada na melhor posição, que foi armazenada.

Na linha 67, na função será chamada MinimaxValue, e na linha 03 será guardado a referência do jogador atual (jogador artificial X: MAX ou jogador humano O: MIN). Na linha 05, será verificado se algum jogador (e qual) ganhará a partida com base no estado dela neste momento. Se ninguém ganhar retorna nulo. Na linha 07, verifica se o jogo chegou em um estado terminal: se alcançou alguma profundidade defina, ou se existe algum vencedor. Nas linhas 09 e 11 indicam respectivamente, se o jogador artificial ganhou e retorna um valor positivo (1). Caso contrário, nas linhas 13 e 15 indicam respectivamente, se o jogador humano ganhou e retorna um valor negativo (-1). Se ninguém ganhou, estado terminal de empate, na linha 17 retorna 0. Na linha 20, verificará se vai utilizar jogadas do jogador MAX, senão o jogador MIN.

Na linha 22 está declarando uma variável pontuação para armazenar a maior pontuação das jogadas daqui em diante. Na linha 23, inicia o loop para cada espaço na mesa. Na linha 25, se a posição da mesa estiver vazia. Na linha 27, a posição vazia recebe o símbolo O do jogador atual (que é o jogador artificial: MAX). Na linha 28, será chamado a função MinimaxValue, passando o objeto do jogo, um campo indicando que será utilizado o jogador MIN, a profundidade da árvore - 1 e assim retornando uma pontuação. Na linha 29, a posição novamente é resetada com um símbolo vazio, permitindo no loop seguinte tentar novas possibilidades de jogadas. Entre as linhas 30 e 33 é verificado se a pontuação retornada é maior que a pontuação atual armazenada: se for, então atualiza-la. Na linha 54 é retornado esta pontuação.

Na linha 40 está declarando uma variável pontuação para armazenar a menor pontuação das jogadas daqui em diante. Na linha 41, inicia o loop para cada espaço na mesa. Na linha 43, se a posição da mesa estiver vazia. Na linha 45, a posição vazia recebe o símbolo X do jogador atual (que é o jogador humano: MIN). Na linha 46, será chamado a função MinimaxValue, passando o objeto do jogo, um campo indicando que será utilizado o jogador MAX, a profundidade da árvore - 1 e assim retornando uma pontuação. Na linha 47, a posição novamente é resetada com um símbolo vazio, permitindo no loop seguinte tentar novas possibilidades de jogadas. Entre as linhas 48 e 51 é verificado se a pontuação retornada é menor que a pontuação atual armazenada: se for, então atualiza-la. Na linha 36 é retornado esta pontuação.

No final do processo, a variável posição terá o valor da posição da mesa com a melhor jogada, e com isso o jogador artificial, fará sua jogada nesta posição.

4.4 Considerações sobre o capítulo

Este capítulo encerra o presente trabalho tendo abordado sobre o jogo de cartas Triple Triad, como regras e informações gerais de como o jogo funciona, como também o código fonte descrito e comentado com o algoritmo Minimax aplicado. E também o código fonte descrito e comentado do jogo da velha com o algoritmo Minimax aplicado.

5 CONCLUSÃO

A grande importância na área da Inteligência Artificial e com o seu crescimento ao longo dos anos, graças aos vários pesquisadores, ajudou a resolver muitos problemas complexos, por exemplo, tomada de decisão. A evolução dos Jogos Digitais se deu graças à evolução do hardware e permitiu que os algoritmos fossem executados dentro de um espaço de tempo viável. A estrutura de dados do tipo árvore genérica, permite buscar informações nos nós e trabalhar com essas informações para resolver problemas.

Com o Algoritmo Minimax através da árvore Minimax, é possível resolver problemas complexos de decisão de jogadas, como por exemplo, jogadas em um jogo da velha representados pelos jogadores MAX e MIN, que devem maximizar e minimizar, respectivamente, o valor de utilidade que representará para qual jogada o jogador da decisão terá que seguir, com base na estrutura da árvore do Minimax. O algoritmo Minimax é recursivo e dependendo da quantidade de jogadas que será calculada, pode afetar o desempenho do algoritmo. A importância da função de utilidade do algoritmo Minimax e ela que definirá o comportamento do jogador artificial, contra o jogador adversário.

O jogo Triple Triad é um jogo de cartas, e possui características (dois jogadores, competitivamente, alternando entre turnos, sendo que, se um vencer o outro perde e vice-versa ou empata) para que o algoritmo Minimax fosse aplicado e resolver o problema abordado neste trabalho, criando um jogador artificial capaz de fazer jogadas para vencer o adversário.

Como trabalho futuro, sugere-se a implementação do corte alfa-beta no algoritmo Minimax, descartando jogadas que não devem ser analisadas e melhorar significantemente desempenho do algoritmo. Também as outras regras do jogo Triple Triad (Elemental, Plus, Same, Same Wall e Combo) que influenciam na decisão das jogadas, fazendo o jogador artificial capaz de jogar com essas regras parar vencer o adversário, deixando o jogo ainda mais interessante e desafiador para o jogador humano.

6 REFERÊNCIAS

ANENE, Francine; MAYEFSKY, Eric; SIROTA, Marina. **Algorithms:** Alpha-Beta Pruning. [s.d.]. Disponível em: <http://web.stanford.edu/~msirota/soco/alphabeta.html>. Acesso em: 13 outubro 2014.

BARTON, Matt; LOGUIDICE, Bill. **Vintage Games:** An Insider Look at the History of Grand Theft Auto, Super Mario, and the Most Influential Games of All Time. Oxford: Focal Press, 2009.

BERENS, Kate; HOWARD, Geoff. **The Rough Guide to Videogames 1**. London: Rough Guides, 2004.

DIGITAL GAMES. Digital Games: History and Genres. **Digital Games**. [s.d.]. <http://www.tlu.ee/imke/gameinteractions/digital%20games.pdf>. Acesso em: 05 agosto 2014.

DONOVAN, Tristan. **Replay:** The History of Video Games. Lewes: Yellow Ant, 2010.

EL-BASUNY, Tarek Helmy. **Principles to Artificial Intelligent**. [s.d.]. Disponível em: <http://opencourseware.kfupm.edu.sa/colleges/ccse/ics/ics381/files/2_Lectures%2021-22-Games%20and%20Adversarial%20Search-Ch.6.pdf>. Acesso em: 19 outubro 2014.

FREEIMAGE. **FreeImage**. [s.d.]. Disponível em: <http://freeimage.sourceforge.net>. Acesso em: 08 agosto 2014.

FREETYPE. **FreeType**. [s.d.]. Disponível em: <http://www.freetype.org/index.html>. Acesso em: 08 agosto 2014.

GUNAWARDENA, Ananda. **Tree Data Structure**. 2007. Disponível em: <http://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson4_1.htm >. Acesso em: 07 setembro 2014.

KENT, Steven. **The Ultimate History of Video Games:** From Pong to Pokemon--The Story Behind the Craze That Touched Our Lives and Changed the World. New York: Three Rivers Press, 2001.

KUSHNER, David. **Masters of Doom:** How Two Guys Created an Empire and Transformed Pop Culture. New York: Random House, 2004.

LANZI, Pier Luca. **A Short History of Videogames:** Videogame Design and Programming. [s.d.]. Disponível em: <http://www.pierlucalanzi.net/wp-content/teaching/vdp/VDP2011-01-History-Short.pdf>. Acesso em: 05 agosto 2014.

LAVENDER, Greg. **Algorithms and Data Structures using C++**. 1996. Disponível em: <http://www.cs.utexas.edu/users/lavender/courses/EE360C/lectures/lecture-23.pdf >. Acesso em: 07 setembro 2014.

LIN, Yosen. **Game Trees**. 2003. Disponível em: <http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html>. Acesso em: 14 outubro 2014.

MICROSOFT. DirectX Software Development Kit. **Microsoft**. [s.d.]. Disponível em: <http://www.microsoft.com/en-us/download/details.aspx?id=6812>. Acesso em: 08 agosto 2014.

NILSSON, Nils J. **The Quest for Artificial Intelligence.** New York: Cambridge University Press, 2009.

NORVIG, Peter; RUSSELL, Stuart. **Artificial Intelligence:** A Modern Approach. 3. ed. New Jersey: Prentice Hall, 2009.

OPENAL. **OpenAL**. [s.d.]. Disponível em: <http://openal.org>. Acesso em: 08 agosto 2014.

PINTO, Paulo. **Introducing the Min-Max Algorithm**. 2002. Disponível em: <http://www.progtools.org/games/tutorials/ai_contest/minmax_contest.pdf>. Acesso em: 27 agosto 2014.

ROSEN, Bruce. **Minimax with Alpha Beta Pruning**. [s.d.]. Disponível em: <http://cs.ucla.edu/~rosen/161/notes/alphabeta.html>. Acesso em: 13 outubro 2014.

SQUARESOFT. **Squaresoft**. 1999. Disponível em: <http://www.final-fantasyviii.com/>. Acesso em: 07 dezembro 2014.

SUH, Eric. **MiniMax Game Trees**. Disponível em: <http://www.cprogramming.com/tutorial/AI/minimaxtree1.html>. Acesso em: 27 agosto 2014.

UNIVERSITY OF PENNSYLVANIA. Tree Searching. **University of Pennsylvania**. Disponível em: <http://www.cis.upenn.edu/~matuszek/cit594-2012/Lectures/31-tree-searching.ppt>. Acesso em: 07 setembro 2014.

VORBIS. **Vorbis**. [s.d.]. Disponível em: <http://www.freetype.org/index.html>. Acesso em: 08 agosto 2014.

WARWICK, Kevin. **Artificial Intelligence:** The Basics. New York: Routledge, 2011.

WU, Dekai. **Depth-First Search**. 2005. Disponível em: <http://www.cse.ust.hk/~dekai/271/notes/L06/L06.pdf>. Acesso em: 07 setembro 2014.