UNIVERSIDADE FEDERAL DO PAMPA
CIÊNCIA DA COMPUTAÇÃO

LARAÍNE RAMOS DOS ANJOS

ALGORITMOS DE BUSCA DE CAMINHO EM *TILE GAMES*

Trabalho de Conclusão de Curso

Alegrete
2013

LARAÍNE RAMOS DOS ANJOS

ALGORITMOS DE BUSCA DE CAMINHO EM *TILE GAMES*

Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de bacharel em Ciência da Computação na Universidade Federal do Pampa.

Orientador: Prof. Dr. Fábio N. Kepler

Alegrete/2013

LARAÍNE RAMOS DOS ANJOS

ALGORITMOS DE BUSCA DE CAMINHO EM *TILE GAMES*

Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de bacharel em Ciência da Computação na Universidade Federal do Pampa.

Trabalho apresentado e aprovado em: 04 de outubro de 2013.

Banca examinadora:

_____

Prof. Dr. Fábio N. Kepler

Orientador

Ciência da Computação – UNIPAMPA

_____

Profª. Aline Vieira de Mello

Ciência da Computação – UNIPAMPA

_____

Prof. Dr. Daniel Welfer

Ciência da Computação – UNIPAMPA

Dedico este trabalho a Inamar Ramos dos Anjos, minha mãe, modelo de mulher forte, perseverante e de caráter inabalável.

AGRADECIMENTO

Ao Grande Ser que governa o universo, independente de religião ou credo e que, sempre se fez presente em todos os momentos de minha vida.

À minha família, por sua capacidade de me apoiar, acreditar e investir em mim. Mãe, seu cuidado e dedicação me deram, em muitos momentos, a esperança para continuar. Pai, sua presença, ainda que tímida, foi, como sempre, a segurança e certeza de que não estou sozinha nessa caminhada.

Ao Giovane, que acompanhou de perto esses longos anos de formação. Com você aprendi e aprendo a cada dia, meu amor. Obrigada pelo carinho, amor, paciência e por sua capacidade de me trazer paz na correria de cada semestre. Por me amparar quando achei que as minhas forças não seriam suficientes.

À minha madrinha e segunda mãe, Nilza, por me proporcionar uma segunda casa onde sempre me senti acolhida e por seus braços sempre abertos para me receber.

Aos colegas e professores, sem os quais, nada seria igual e tão especial.

A todos aqueles que de alguma forma estiveram e estão próximos de mim, fazendo cada segundo desta vida valer cada vez mais a pena.

"Obrigado a todas as pessoas que contribuíram para meu sucesso e para meu crescimento como pessoa. Sou o resultado da confiança e da força de cada um de vocês."

Augusto Branco

RESUMO

Uma das principais características que se busca em um jogo é a representação da realidade. Não basta apenas os movimentos do próprio jogador, é necessário que os outros personagens ajam de acordo com o que se espera deles se o ambiente do jogo fosse o mundo real. Para isto são usados vários algoritmos que ajudam na "percepção" do ambiente, movimentação e realização de objetivos por parte dos NPCs (*Non-Player Character*). É importante salientar que esta "expectativa" independe da plataforma em que o jogo está instalado, ou seja, espera-se que um personagem de um jogo de *browser* seja tão inteligente quanto aquele que faz parte de uma determinada plataforma exclusiva para jogos e com hardware próprio para esta atividade. O objetivo deste trabalho é aplicar algoritmos de busca de caminho em jogos que utilizem plataformas com recursos limitados, como *tablets* e celulares, e assim obter uma análise de quais são mais adequados, seja em função de tempo, memória ou otimalidade. Os protótipos desenvolvidos para testar esses algoritmos são construídos a partir do método *Tile*. Esse método visa diminuir a quantidade de memória utilizada pelo jogo através da divisão de imagens grandes em várias pequenas partes, que são replicadas quando necessário para reproduzir a imagem original.

Palavras-chave: inteligência artificial, busca de caminho, *tile games.*

ABSTRACT

One of the most desired features in a game is a representation of reality. It is not enough to just move the player, it is necessary that the other characters act according to what is expected of them if the game environment was the real world. For this, various algorithms are used to help the "perception" of the environment, drive and achievement of goals by the NPCs (Non Player Characters). It is noteworthy that this "expectation" is independent of the platform on which the game is installed, i.e., it is expected that a character from a browser game is as smart as one who belongs to a particular platform exclusive for games and with hardware suitable for this activity. The objective of this work is to apply pathfinding algorithms in games that use platforms with limited resources, such as tablets and phones, and thus obtain an analysis of which are most suitable, either in terms of time, memory, or optimality. The prototypes developed to test these algorithms will be built from the Tile method.. This method aims to reduce the amount of memory used by the game, by splitting large images into several smaller parts, which are replicated as needed to reproduce the original image.

Keywords: *artificial intelligence*, *pathfinding*, *tile games.*

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

2D – Duas Dimensões

AS2 – Action Script 2

FPS – *First-Person Shooter*

IA – Inteligência Artificial

IBM - *International Business Machines*

NPC – *Non-player Character*

RTS - *Real Time Strategy*

TCC - Trabalho de Conclusão de Curso

SUMÁRIO

1 INTRODUÇÃO

O início dos jogos eletrônicos remonta de meados dos anos 50, começando em computadores cujo hardware não era voltado para este tipo de atividade e, posteriormente desenvolvidos para consoles voltados apenas para esta finalidade. Baseado na limitação destes hardwares, como por exemplo a memória limitada e a pouca velocidade dos processadores, era necessário o desenvolvimento de códigos com poucas linhas e fáceis de compilar. Entretanto, este mesmo jogo deveria prender a atenção do jogador e tentar ao máximo ser fiel ao mundo real ou ambiente que se desejava representar, o que é chamado de ambientação.

A ambientação ou imersão do jogador ao jogo ocorre baseada em diversos fatores como: gráficos e texturas dos cenários, relação com os outros personagens (controlados por outros jogadores ou pelo próprio jogo), sons, história, identificação com o personagem principal, etc.

Uma das formas de adequar as expectativas do jogador à realidade do hardware disponível na época e ainda atentar para a qualidade gráfica do jogo foi o método *Tile* (azulejo, em uma tradução livre), que visava diminuir o tamanho das imagens utilizadas nos jogos, de forma a dar ao produto final duas características: tamanho reduzido e, ainda assim, um detalhamento de imagem significativo.

Outra parte importante da ambientação é a relação com os outros personagens do jogo. Neste contexto, inclui-se a movimentação coerente, ou seja, um jogo perde totalmente a credibilidade se um personagem, ao procurar um caminho, fica preso entre duas pedras, ou anda em círculos, por exemplo.

Para que um personagem se desloque de um ponto A até um ponto B em um ambiente, é necessário que ele consiga definir um conjunto de ações/movimentos que o façam chegar até seu objetivo. Mas não basta apenas isto, é preciso ainda, que este caminho seja o melhor possível baseado nas próprias definições do ambiente e do jogo. Os algoritmos que indicam este caminho são chamados de algoritmos de busca de caminho.

A definição de qual algoritmo de busca de caminho é melhor depende do que é esperado pelo jogo analisado. Por exemplo, para alguns jogos o algoritmo melhor é aquele que acha o caminho mais rápido, independente se o caminho encontrado é o mais curto. Outras vezes, precisa-se que o caminho tenha o menor número de passos possível, independente da quantia de memória que o algoritmo utiliza.

Independente da plataforma em que o jogo for executado, seja em um console de última geração ou em um *browser,* espera-se que o jogador tenha a mesma experiência, identificação e ambientação. Entretanto, da mesma forma que no início de sua história, quando os jogos deviam ser executados em *hardwares* limitados, hoje há plataformas que são utilizadas para estes fins, mas que não estão preparadas para um grande processamento e armazenamento. Neste grupo, além de browsers, podemos incluir *tablets* e celulares. Baseados nas informações anteriores, podemos afirmar que o método *Tile* pode ser utilizado na programação de jogos para estas plataformas e que sua forma de estruturação pode abrir um leque de opções para a movimentação dos personagens neste novo cenário, nos propiciando a melhoria de duas fases importantes da ambientação do jogador: qualidade gráfica e relação com outros personagens. O método também reflete na complexidade das ações dentro da lógica do jogo, uma vez que provê uma grade como cenário (importante para os algoritmos de busca de caminho) e trata cada pedaço de imagem como um objeto que tem suas propriedades facilmente acessadas (fato que facilita a detecção de colisões com o cenário, terrenos que não podem ser percorridos e/ou propriedades físicas, como atrito).

Tomando os celulares como exemplo, apesar da evolução rápida de seu hardware (o que permite que executem algoritmos cada vez mais complexos), não podemos descartar os dispositivos mais antigos, com recursos limitados, pois ainda são responsáveis por uma fatia grande do mercado consumidor. Mesmo nestes dispositivos é importante o estudo do melhor uso da Inteligência Artificial (IA), independentemente de quais tipos de algoritmos estão sendo usandos para simular essa inteligência, ou qual é a abordagem mais adequada. Isso se deve ao fato de que, muitas vezes o jogador acaba por ter como adversário a própria máquina, seja pelo fato de o telefone não possuir capacidade de conectividade ou por escolha do próprio usuário. Isso se deve ao fato de que, muitas vezes o jogador acaba por ter como adversário a própria máquina, seja pelo fato dela não possuir capacidade de conectividade ou por escolha do próprio usuário. Isso descartaria o uso da programação em nuvem, por exemplo, deixando a carga de execução dos algoritmos apenas na máquina do jogador. Nesse caso, a capacidade do desafio proposto pela inteligência artificial envolvida no jogo deixa de ser trivial e passa a ser parte central da experiência do usuário com o software.

1.1. Objetivos

O objetivo deste trabalho é explorar técnicas da área de Inteligência Artificial para movimentar um NPC (*Non-Player Character)* de forma inteligente em ambientes baseados

em *Tiles*. Como já citado, este método é utilizado frequentemente em *hardware* com pouca memória e processamento reduzido. Pretende-se que esta movimentação possa ser desenvolvida nestas condições e que, ainda assim, apresente bons resultados.

Para que estes algoritmos possam ser testados, foi desenvolvido um jogo baseado em *Tiles*. As propriedades dos *Tiles* oferecerão aos algoritmos de busca de caminho as informações necessárias para a movimentação do personagem e escolha do melhor caminho a seguir. Estes algoritmos serão os responsáveis pela adaptação inteligente do agente ao ambiente e consequente finalização de suas tarefas.

1.2. Organização deste trabalho

Este trabalho está organizado da seguinte maneira: No Capítulo 2 descreve-se os fundamentos teóricos abordados: inteligência computacional, busca de caminho em Inteligência Artificial e o método *Tile*. Nesse capítulo também são abordados uma série de trabalhos relacionados e o estado da arte do estudo proposto. No Capítulo 3 faz-se a explanação da metodologia abordada durante o período de desenvolvimento deste trabalho. No Capítulo 4 apresenta-se o *Game Design* dos protótipos criados durante a etapa de desenvolvimento. No Capítulo 5 encontram-se os resultados obtidos, assim como as medidas de comparação utilizadas e a discussão dos resultados encontrados. No Capítulo 6 apresentam-se as considerações finais, referentes aos dados apresentados no capítulo anterior.

2 FUNDAMENTOS

De acordo com Caillois (1986) o jogo é uma atividade livre e voluntária, capaz de trazer diversão, onde predomina a incerteza e o caráter improdutivo. Já Negrine (1994) diz que jogar não é apenas uma atividade e sim uma atitude que emana uma vivência de sentimentos e sensações que nos fazem desvendar significados e tomar decisões.

Entre as décadas de 60 a 70, os jogos eletrônicos utilizavam apenas a ideia de dois jogadores (*versus mode*), mas com o aumento da comercialização a indústria percebeu que a opção para apenas um jogador poderia aumentar suas vendas e lucros. Mas isso trouxe à tona uma importante questão: como implementar adversários virtuais que agissem inteligentemente, de forma a manter o desafio e a diversão do jogador ao jogar sozinho?

No meio acadêmico já existiam técnicas que eram capazes de oferecer a entidades virtuais modos de autonomia, raciocínio e lógica para resolução de problemas. Essas técnicas visavam a simulação do comportamento humano frente a diversas situações.

Com o surgimento do modo *single-player* (apenas um jogador), essas técnicas passaram a ser aplicadas nos jogos, como forma de fazer os personagens controlados pela máquina serem capazes de desenvolver uma complexa rotina de tomada de decisões, como escolher o melhor caminho, tomar uma decisão baseada em uma ação do jogador, etc. Esse comportamento faz com que o jogador as visualize como entidades inteligentes (agente inteligente), aumentando a imersão e diversão do jogo.

A tabela abaixo esquematiza o uso da I.A em alguns jogos eletrônicos:

Tabela 1 - Histórico da utilização de I.A em jogos

Nível de I.A	Aplicação	Ano
Sem I.A	Space War – primeiro jogo de computador. Escrito para o minicomputador PDP-1. Requer dois jogadores	1962
	Pong- versão eletrônica do tênis de mesa	1972
Padrões	Qwak, com padrões de movimentação (o jogador precisava atirar em alvos móveis)	1974
	Gun Fight – personagens com movimentação aleatória	1975
	Space Invaders – inimigos são padronizados, mas atiram de volta. Considerado o primeiro jogo "clássico", com níveis, score, controles simples e e dificuldade crescente no decorrer do jogo	1978
	Pac-man – fantasmas com movimento padronizado, mas fantasma possui uma personalidade	1980
	Um microcomputador vence um jogador profissional de xadrez pela primeira vez	

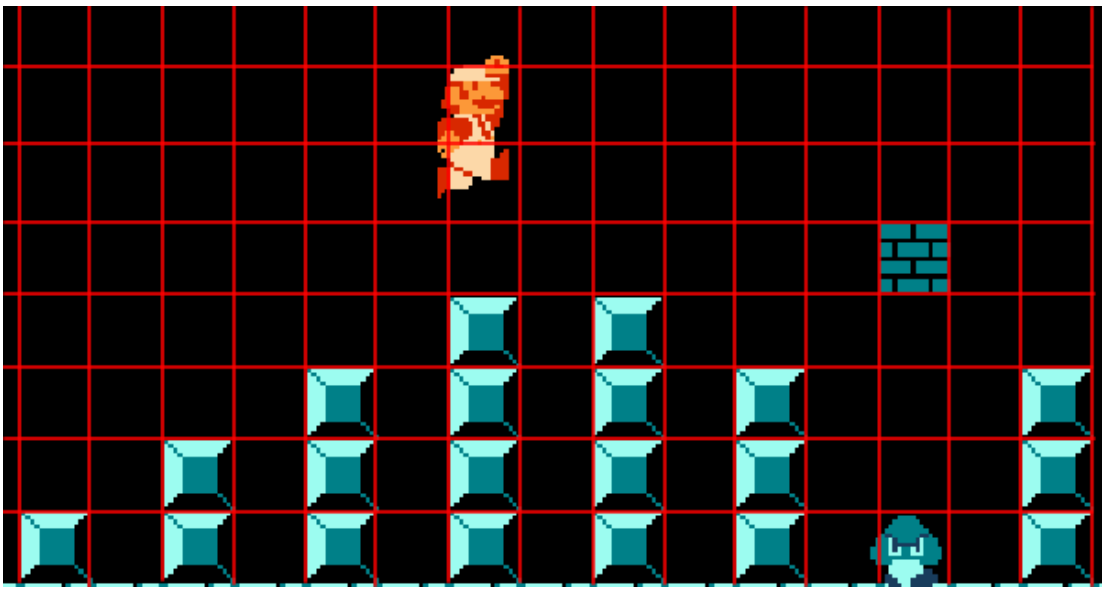
	Karate Champ(Data East, 1984) – um dos primeiros jogos de luta um contra um, com o computador como adversário	1984
FSM's	Herzog Zwei (TecnoSoft, 1990) – O primeiro RTS a surgir, o mundo experimenta pela primeira vez uma péssima implementação de algoritmo de busca de caminho, sendo considerada previsível após algum tempo de jogo.	1990
	Doom, Wolfenstein 3D (Id Software). – Primeiro jogo de tiro em primeira pessoa	1992
	Doom (Id Software, 1993)	1993
Várias Técnicas	BattleCruiser: 3000AD (Take Two Software,1996) – Primeiro uso de redes neurais em um jogo comercial	1996
	Deep Blue derrota o atual campeão de xadrez Gary Kasparov	1997
	Half Life - A inteligência artificial em jogos encontra-se em seu auge. O jogo faz grande uso de linguagens de script.	1998
	Black&White (Lionhead Studios, 2001) – Utiliza criaturas que usam aprendizado por reforço e observação	2001

Baseada em Santana, 2006 e KISHIMOTO, 2004, com informações extras.

2.1. Método *Tile*

Na época em que os primeiros jogos começaram a ser comercializados não era possível usar gráficos detalhados, pois estes eram muito grandes e demoravam a ser interpretados. Então, para preencher esta lacuna, foi desenvolvido o método *Tile* (azulejo, em uma tradução livre). Esse método divide a imagem em várias partes que são chamadas de *Tiles*. Na Ilustração 1, encontra-se a divisão em *Tiles* de um dos cenários do jogo *Mario Bros.*

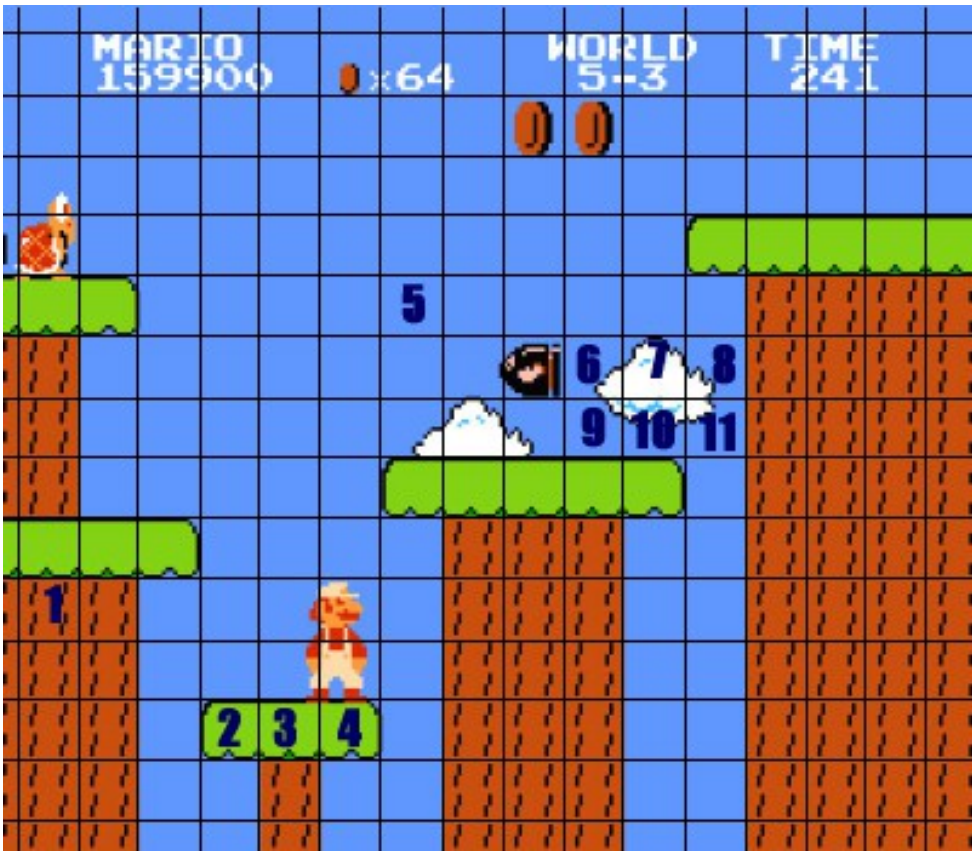
Ilustração 1 - Divisão de uma imagem em partes iguais



Fonte: Screenshot do jogo Super Mario Bros (Nintendo, 1985)

Após a divisão sistemática da imagem (conforme a abordagem, a imagem pode ser dividida utilizando-se uma ou diversas formas geométricas para fazer o mapeamento de suas partes, com tamanho padrão ou não), percebe-se que geralmente um mesmo *Tile* pode se repetir em várias posições diferentes na imagem inicial. Ao verificar isto, deve-se então separar todos os *Tiles* diferentes e mapear os lugares onde cada um se repete, utilizando-se uma matriz como representação da imagem e outras estruturas de dados (geralmente vetores) para o mapeamento do lugar onde cada *Tile* deverá ser replicado. Na Ilustração 2, pode-se perceber os diferentes *Tiles* que formam um cenário do jogo *Mario Bros*.

Ilustração 2 - Identificação de *tiles* que se repetem em uma imagem



Fonte: Screenshot do jogo Super Mario Bros (Nintendo, 1985)

A partir disto, é possível perceber que não é mais necessário que toda a imagem seja carregada na execução do programa e sim, os principais *Tiles*. Estes são replicados e encaixados para formar a imagem pretendida.

Com este método se pode diminuir significativamente a quantidade de memória utilizada com os recursos gráficos do jogo. É possível ainda, aliada às técnicas atuais de programação, identificar cada *Tile* como um objeto, definir suas propriedades e como os

agentes do mundo interagem com ele. Esta adaptabilidade do método faz com que seja usado até hoje em produções de jogos onde se precisa de economia de recursos, mas não se quer perder muitas qualidades gráficas.

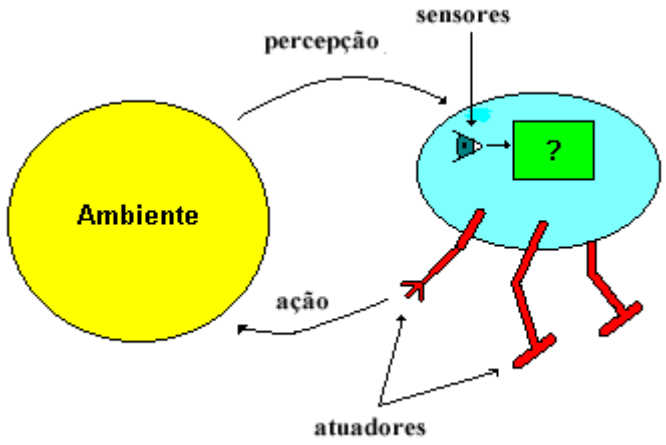
2.2. Agentes e Ambientes

Segundo Russel e Norving (2003, 1995) **agentes** são todos os elementos capazes de perceber o ambiente onde estão inseridos por meio de sensores próprios e agir neste ambiente através de atuadores.

Se tomarmos um robô como exemplo de agente, podemos considerar que, se ele tem câmeras no lugar dos olhos, estas são sensores de percepção do ambiente. Ou seja, através delas, o robô pode ter uma noção de onde pode se movimentar, dos obstáculos e traçar vários cálculos a partir destas informações, proporcionando uma série de escolhas entre várias ações possíveis. Para executar um movimento baseado nesses cálculos, seria preciso uma série de motores, rodas ou algo do gênero, ou seja, estes seriam os atuadores.

Um **agente racional** ou inteligente é aquele que dadas várias entradas (percepções fornecidas pelos seus sensores) toma a melhor ação baseada nos seus conhecimentos internos e nas entradas que recebe, de forma a ter sua melhor performance naquela situação. A Ilustração 3 mostra um esquema do funcionamento de um agente inteligente.

Ilustração 3 - Esquema de um agente inteligente



Fonte: Imagem baseada em Russel & Norvig (2003)

Os **ambientes** em que os agentes são inseridos podem ser classificados de diversas formas, segundo as características de suas mudanças, previsão ou sua relação com o agente.

Um **ambiente completamente observável** é aquele que a todo instante os sensores do agente conseguem perceber o estado completo do ambiente.

Se o próximo estado do ambiente é totalmente previsível baseado no estado atual do ambiente e nas ações do agente, este **ambiente** é **determinístico**, senão é **estocástico**.

Consideramos um episódio o conjunto da percepção do agente e de uma única ação sua. Em um **ambiente episódico**, a decisão do agente no atual episódio não tem qualquer efeito nos próximos episódios que se seguirão. Já no caso de um **ambiente sequencial**, a decisão atual pode afetar todas as decisões futuras.

Se o ambiente pode mudar enquanto o agente está decidindo suas ações, dizemos que é um **ambiente dinâmico**, senão, é **estático**.

Um **ambiente discreto** é aquele que tem um conjunto finito de estados, ações e percepções. Se, no entanto, este conjunto de ações, estados e percepções for tão grande e variado, que pode ser comparado ao infinito, dizemos que o **ambiente** é **contínuo**.

2.3. Comportamento de NPCs

Podemos dizer que, atualmente, uma das principais partes observadas em um jogo é a interatividade e a inteligência dos personagens controlados pelo computador. Isso se deve ao fato desta trazer coerência e a consequente identificação do jogador com o jogo. Esta identificação é esperada independente da plataforma em que o jogo está sendo executado, ou seja, mesmo que o jogo esteja rodando em um *browser* ao invés de em um console próprio, espera-se que as atitudes dos NPCs sejam coerentes.

Espera-se que um NPC seja capaz de perceber o seu ambiente e interagir com ele de forma muito próxima às atitudes humanas. Como exemplo, um ser humano ao ver um labirinto, tenta traçar uma série de atitudes de forma a chegar ao final do mesmo, ou seja, toma uma sequência de ações que o levam a um objetivo. Da mesma forma, em um jogo, espera-se que os personagens controlados pelo computador sejam capazes de se deslocar pelo ambiente o mais naturalmente possível. Não seria tolerável que o personagem ficasse se debatendo entre três paredes do labirinto, por exemplo.

Então, atualmente, uma das partes mais importantes na criação de um jogo é a inteligência artificial envolvida nas ações dos agentes controlados pelo computador. Sobre

este aspecto, pode-se citar a necessidade de criar algoritmos que controlem a navegação dos agentes pelo ambiente em que estão inseridos. Algoritmos de busca de caminho, também chamados de algoritmos de navegação, devem ser desenvolvidos para que os agentes sejam capazes de desviar de objetos presentes no seu caminho e consigam chegar ao seu objetivo.

A maioria dos algoritmos de busca de caminho utiliza uma visão do mundo baseada em uma matriz ou *grid*, da mesma forma como o método *Tile* mapeia o ambiente para distribuição das partes da imagem. É possível perceber que juntando os dois, já temos os principais pilares de um jogo: um método para inclusão de imagens com economia de memória e alguns algoritmos que podem fazer os agentes deslocarem-se por este mundo baseado em *Tiles*.

Entretanto, como no início, o método *Tile* é usado em *hardwares* que não foram inicialmente desenvolvidos para o uso em jogos. Por exemplo, há vários celulares e mais atualmente, *tablets* que podem executar jogos, mas sabemos que a quantidade de memória e a capacidade dos processadores desses equipamentos não é tão desenvolvida quanto em um computador pessoal ou um notebook. Então, temos duas necessidades divergentes: os usuários querem que o jogo execute e se comporte o melhor possível (ou seja, mais próximo à execução em um computador), mas o *hardware* não dá sustentação a algoritmos mais poderosos.

Na maioria das vezes opta-se por um algoritmo de navegação mais simples, como o uso de movimentos aleatórios ou um *script* fixo (padrões de movimento e atuação pré-definidos*)*. A primeira opção, não deixa o jogo repetitivo, mas pode dar a impressão de falsa inteligência, por exemplo, se o agente bate em uma parede e simplesmente "sorteia" para que lado vai em seguida; pode ser que após poucos movimentos volte a bater na mesma parede. A segunda faz com que o agente pareça mais inteligente, mas com sucessivas execuções, fica claro ao usuário que o caminho do agente é pré-determinado.

Nos dois casos acima a ambientação é prejudicada. No primeiro caso, ao perceber que os agentes não tem inteligência condizente com a realidade, o jogador pode perder o interesse no jogo. E no segundo caso, após sucessivas execuções, a partir do momento que o jogador pode "prever" o comportamento do NPC, o jogo pode se tornar "muito fácil", tirando boa parte da atratividade do mesmo.

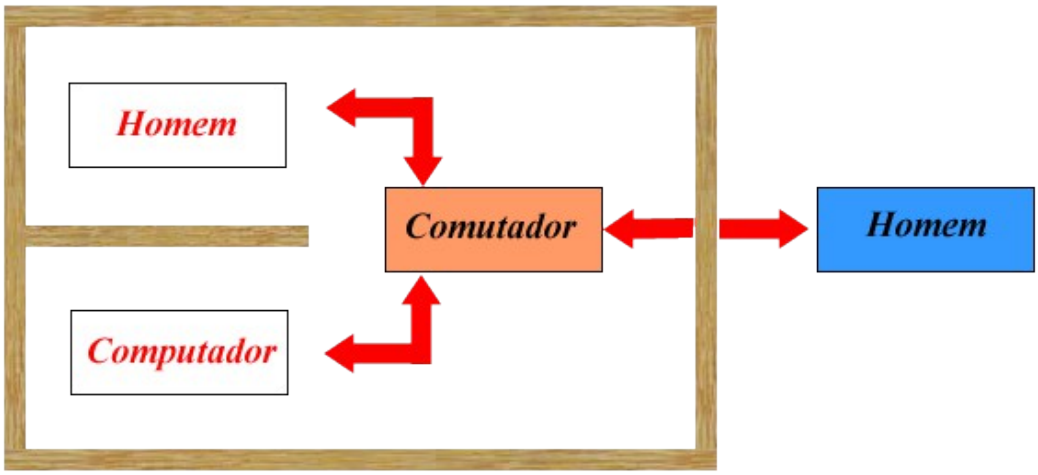
2.4. Inteligência e Teste de Turing

Em 1950, Alan Turing publicou um artigo na revista filosófica *Mind* sob o título **"Computing Machine and Intelligence"**. Este artigo descrevia pela primeira vez o que hoje é conhecido como **Teste de Turing**, o qual tinha como objetivo determinar se um programa é ou não inteligente. Segundo ele, em vez de perguntar se um computador pode pensar, deveríamos verificar se ele consegue passar por um teste de inteligência comportamental. O teste de Turing é baseado na modificação do **Jogo de Imitação** (*Imitation game*, em inglês) que funciona da seguinte forma:

Há três participantes no jogo: um homem (1), uma mulher (2) e um interrogador (3). O interrogador não tem contato direto com os outros participantes e não sabe qual o sexo dos mesmos. A comunicação entre as entidades é feita de maneira indireta e o interrogador deverá identificar através do diálogo se (1) é homem e (2) é mulher ou vice-versa.

Na formulação de seu teste, Turing substituiu um dos participantes por uma máquina programada para imitar o comportamento humano. Ao final do teste o interrogador deverá identificar qual dos participantes é humano e qual é máquina. Se no final do experimento, o interrogador não conseguir identificar quem é humano, conclui-se que a máquina é inteligente.

Ilustração 4 - Esquema do teste de Turing



Fonte: Imagem baseada na descrição do método

Até os dias de hoje nenhuma máquina conseguiu passar no Teste de Turing completo. Entretanto algumas máquinas conseguiram obter um desempenho aceitável em variações mais brandas do teste, onde o atributo mental do entendimento não era cobrado.

2.5. Negação de Inteligência, segundo Turing

Inaptidão

Segundo Santana (2006), há várias ações que as máquinas não conseguiriam executar, como ser amável, se apaixonar, ser empático, etc. Apesar de hoje existirem máquinas capazes de realizar tarefas que exigem compreensão e perspicácia humana, não se pode dizer que eles possuem essas qualidades. Por exemplo, um software que faz um diagnóstico médico, pode acertar, mas não se pode dizer que ele tem noção de como o paciente se sente ou que, ele sabe que ao receitar uma injeção o paciente poderá se sentir desconfortável.

Objeção Matemática

Segundo Santana (2006), o teorema da **incompleteza de Gödel** é uma das provas que as máquinas tem inteligência inferior aos humanos, pois indica que as máquinas são sistemas formais limitados pelo teorema da incompleteza. Já os humanos não possuem essa limitação.

Entretanto, essa afirmação tem vários de seus pontos atacados, como por exemplo: um agente não pode se contradizer, então ele não está errado em não conseguir afirmar se uma sentença é verdadeira se a mesma se contradiz.

Outro ponto é que muitos acham que a matemática não pode ser usada como definição de inteligência, fornecendo como exemplo o fato de que os humanos não eram considerados menos inteligentes antes do invento da matemática. Ou seja, uma máquina não pode ser considerada inteligente só porque tem grande domínio da matemática, assim como alguém que não sabe contar não poder ter sua inteligência negada.

Informalidade

Para Santana (2006), segundo Turing um computador não conseguiria reproduzir o comportamento humano visto sua complexidade, pois aquele é regido por sistemas de regras simples e básicas. O filósofo **Hubert Dreyfus** criticava a forma como os computadores eram programados para "supostamente" desenvolver um comportamento humano. Segundo ele, não se pode comparar um enxadrista humano com um computador que "sabe" jogar xadrez, pois

ao contrário do computador, o humano não testa todas as possibilidades de jogadas e sim escolhe uma que segundo sua experiência é melhor para aquela situação (SANTANA, 2006).

Dreyfus já propunha uma ideia que é utilizada hoje para aprendizagem por parte dos agentes, um sistema de aquisição de experiência (rede neural), baseada em um amplo conjunto de regras.

2.6. Inteligência Artificial Fraca

Seu início remonta do início dos próprios computadores, uma vez que a mesma tem como base a visão do computador como um utensílio que auxilie o homem, mas não necessariamente dotado de inteligência. Seus princípios baseiam-se na ideia de que um computador apenas processa dados e não os "entende", uma vez que não tem estruturas emocionais ou psicológicas como os humanos.

Como um dos defensores da IA Fraca, temos o filósofo americano **John Searle**. Segundo ele, a simples realização de um algoritmo, mesmo que bem-sucedido, não significa que houve entendimento por parte do computador. Não se pode provar que ele não apenas seguiu os passos para que foi programado. Defende que um computador não é capaz de verdadeiramente raciocinar e resolver problemas. Não se envolve em problemas éticos, pois nestes casos, o computador apenas aparenta ser capaz de raciocinar devido a sua extensa programação dotada de várias regras que são utilizadas para prever várias situações, efetuado cálculos lógicos a partir dos dados obtidos e apresentando a melhor entrada encontrada.

Um dispositivo dotado de IA Fraca até poderia manter um diálogo com um ser humano, entretanto suas respostas não poderiam fugir das regras pré-definidas do programa. Ele não "entenderia" a conversa, apenas selecionaria a resposta que tem mais probabilidade de estar certa. Por exemplo, considere um programa onde algumas de suas regras são "animais = bonitos; frango = animal". Se um interlocutor fizesse a seguinte afirmação "Prefiro comer frango no jantar.", há grandes possibilidades de o programa responder "Frangos são bonitos", o que seria considerado ilógico para uma conversação inteligente. Obviamente, quanto mais extensas e aprofundadas forem as regras definidas, maior é a "impressão" de inteligência.

2.7. Inteligência Artificial Forte

Para os estudiosos dessa corrente, o funcionamento do cérebro humano não passa de uma complexa rede de operações, assim como um procedimento para um programa. Desse modo, a consciência não seria nada mais que uma consequência do bom funcionamento deste algoritmo. Entretanto, essa complexidade ainda não pode ser reproduzida, mas quando for possível, segundo eles, a máquina poderá não só pensar, como compreender, sentir e etc.

Kasparov (campeão mundial de xadrez, em 1996), depois de jogar com **Deep Blue** (supercomputador criado e programado pela *International Business Machines* – IBM – para jogar xadrez), levantou a seguinte questão: se o computador faz o mesmo movimento que um humano faria por razões completamente diferentes, ele fez um movimento inteligente? A inteligência de uma ação depende de quem (ou do que) a faz?

Segundo **Mavin Minsky**, filósofo e matemático, é totalmente possível atribuir qualidades mentais às máquinas, uma vez que se trata apenas de reproduzir o "algoritmo que comanda o cérebro". Em uma entrevista para a revista *Isto é*, declarou que apenas é uma questão de entender realmente como o cérebro funciona, depois fazer com que os computadores o repitam. O maior obstáculo seria o fato de programarmos os computadores para que não haja erros, entretanto, na natureza, não existe esse medo, ou seja, nosso cérebro não tem receio de errar ao tomar uma decisão.

Outro argumento em xeque nestes casos é os desvios de curso da inteligência. Um jogador de xadrez humano, por exemplo, pode ter seu desempenho afetado se estiver nervoso ou mesmo se sentir intimidado. Isto não acontece com uma máquina. Até que ponto seria interessante reproduzir este comportamento?

Turing ressalta que para uma máquina ser verdadeiramente inteligente ela precisa ter consciência dos seus atos e dos estados mentais pelos quais passou para chegar à resposta.

Por IA forte considera-se algo que desenvolva uma forma de inteligência capaz de raciocinar e resolver problemas, além de possuir autoconsciência e raciocínio intuitivo. Se for possível uma máquina chegar a este nível, várias questões éticas seriam envolvidas, como por exemplo: como lidar com uma inteligência não biológica, mas comparada à humana e como ela se relacionaria conosco? Seria possível computadores inteligentes se recriarem a ponto de ultrapassar os dotes humanos?

Vários filmes e documentários foram desenvolvidos sobre essa abordagem, como exemplo temos a trilogia de Matrix, onde os humanos foram totalmente dominados pelas máquinas, que ultrapassaram a inteligência humana após algumas gerações de máquinas criando outras máquinas.

2.8 Teste do Quarto Chinês

John Searle, em seu *paper* **"Minds, Brains and Programs" (1980)** apresenta um experimento mental conhecido como Quarto Chinês, através do qual tenta invalidar a posição assumida por Alan Turing utilizando o Teste de Turing. Nesse *paper*, Searle levanta a questão: levando em consideração o "sentido humano" de inteligência e consciência, o Teste de Turing é válido e confiável ao afirmar que um sistema que obteve êxito no teste é inteligente?

Normalmente a maioria dos cientistas leva em consideração o resultado obtido no Teste de Turing, entretanto Searle afirma que, em determinado momento o resultado do teste pode ser inválido, tomando uma IA fraca como IA Forte e obtendo uma falsa inteligência.

O experimento proposto por Searle se dá da seguinte forma:

Há um humano (tradutor) isolado em um quarto. O tradutor, que compreende apenas inglês, possui um livro de regras escrito em inglês. Há uma fenda no quarto por onde entram folhas vindas do exterior, todas codificadas com símbolos incompreensíveis por parte do tradutor, entretanto ele pode procurar o símbolo no livro e executar as ações descritas pelas regras definidas para aqueles símbolos. Essas regras, por exemplo, podem mandá-lo escrever outros símbolos no papel e o devolver para o exterior.

Na análise de Searle, se o sistema é focado como um todo, quem o observa de fora tem uma falsa impressão de inteligência (no caso, compreensão do idioma chinês), uma vez que uma folha escrita em chinês é enviada ao quarto e outra no mesmo idioma é devolvida. No entanto, como dado inicial temos que o tradutor não entende chinês, ou seja, não tem compreensão do que está escrito na folha recebida nem naquela que enviou para fora do quarto. Desta forma, o sistema não é inteligente, apenas executa uma rotina pré-determinada. Assim, mesmo que um sistema obtenha aprovação no Teste de Turing, isso não implica que ela tem consciência (no sentido humano) do que estava envolvido no teste e, do mesmo modo, o teste não conseguiria diferenciar uma IA Forte de uma IA fraca.

A questão levantada não é a impossibilidade de uma máquina recriar ações humanas e sim, se um ser humano e um computador encaram da mesma forma certa tarefa. Como exemplo, podemos encarar a situação de abandonar uma pessoa em um território que lhe é completamente estranho e sem qualquer possibilidade de obter ajuda. A única informação que ela tem é a localização do ponto onde está (latitude e longitude) e a localização do avião que pode levá-la para casa. Como contraponto, deixamos um pequeno robô dotado de inteligência artificial com a mesma missão, chegar através de um terreno estranho até um ponto objetivo e com as mesmas informações da pessoa acima. Ambos podem concluir a missão, entretanto, sabe-se que a máquina não passará pelas situações de estresse ou outras atribulações sentimentais pelas quais o humano estará submetido. Ou seja, ambos podem fazer a mesma atividade, mas a experiência por que passam é completamente diferente, ou seja, não há sequer garantias que a máquina "sabe" que está perdida da mesma forma que o humano, o que nos leva a considerar sua inteligência como IA Fraca.

Então, para Searle, o Teste de Turing falha ao considerar como IA Forte o simples fato de uma série de passos serem executados corretamente. Salienta-se que o objetivo do teste do quarto chinês não é indicar a impossibilidade da existência de uma IA Forte, apenas mostra que o teste de Turing não é capaz de identificar uma inteligência consciente, apenas a execução correta de processos humanos.

2.9. Busca de caminho

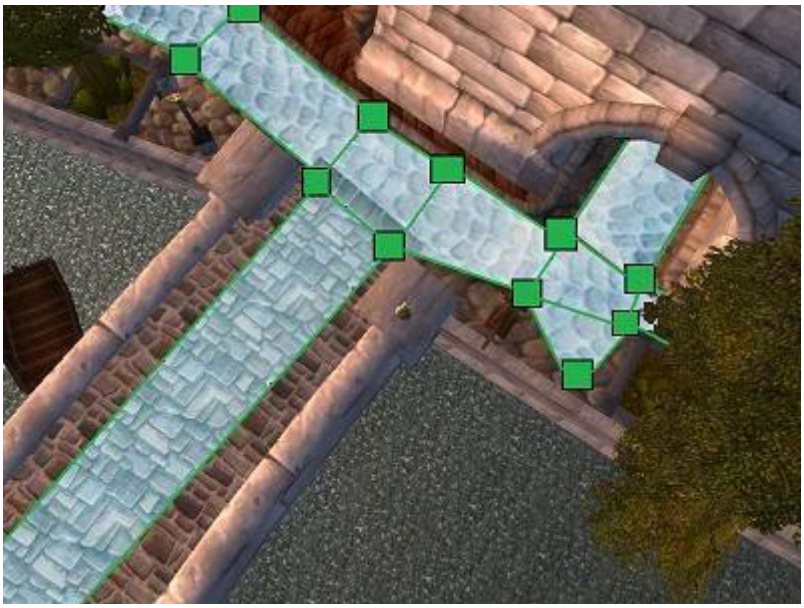
Existem muitos modos de programar inteligência artificial em um jogo. Um deles é utilizar algoritmos de busca de caminho para que os oponentes do jogador percorram determinados caminhos.

Um sistema de navegação de um jogo é responsável por planejar o melhor caminho a seguir por um determinado agente em seu mundo para atingir o seu objetivo. A movimentação do agente, assim como o reconhecimento dos obstáculos e qual o melhor modo de desviá-los é parte integrante do sistema de navegação. Este sistema deve "perceber" o mundo a sua volta para que possa planejar as suas ações.

Existem várias formas de representação de um ambiente para aplicação de um sistema de navegação, dentre eles encontram-se: mapeamento por *NavMesh,* mapeamento por W*aypoints* e mapeamento por *Grid* (Pozzer, 2006).

A primeira técnica consiste em dividir o mundo em uma série de polígonos convexos que são alcançáveis pelo agente. A Ilustração 5, mostra o mapeamento de uma área usando polígonos. Essa técnica geralmente é usada em jogos FPS (First-Person Shooter), como nos jogos Counter Strike e Solider Front, por exemplo.

Ilustração 5 - Mapeamento por *NavMesh*



Fonte: Blog Game/AI[1]

Quando o agente tem de se deslocar de um ponto a outro basta verificar se este ponto está dentro do polígono e seguir em linha reta.

A segunda abordagem cria uma série de pontos que o agente pode alcançar no mundo e em seguida traça um grafo de conexão unindo estes pontos. Então, quando um agente deve mover-se se calcula qual dos pontos do grafo está mais próximo ao ponto final e usa-se as arestas do grafo para fazer a movimentação. Entretanto, este método gera caminhos artificiais e geométricos. Essa técnica é indicada principalmente para jogos FPS onde o ambiente não é muito extenso. A Ilustração 6, mostra um possível mapeamento por *waypoints*, onde podemos perceber que todos os pontos do cenário são alcançáveis através de uma série de retas.

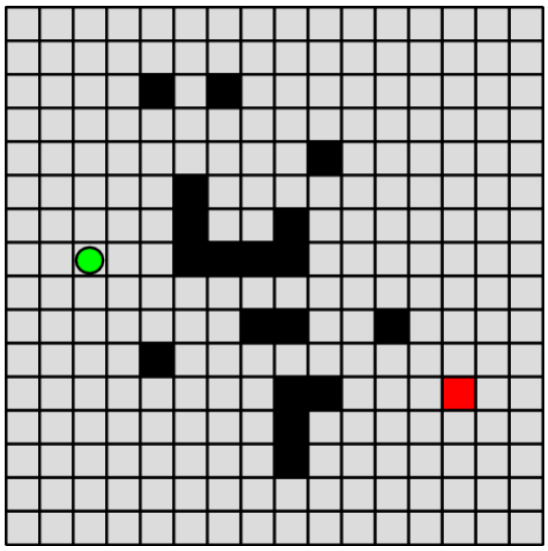
[1] Disponível em <http://www.ai-blog.net/archives/000152.html>

Ilutração 6 – Mapeamento por *WayPoints*

Fonte: Blog Game/AI[2]

A última abordagem faz a representação do mundo na forma de um *grid*. Esta técnica consiste na divisão do ambiente em uma matriz de posições. A Ilustração 7, mostra um cenário representado por um *grid.*

Ilustração 7 – Mapeamento através de um grid



Fonte: Exemplificação do método

[2] Disponível em <http://www.ai-blo.g.net/archives/000152.html>

Esse tipo de representação é muito utilizada para jogos em duas dimensões (2D), ou seja, aqueles onde a posição do agente é definida por um par de coordenas (x,y). A Ilustração 8 mostra um exemplo de jogo 2D, *The Legend of Zelda,* onde a posição do personagem é definida em função de coordenadas na matriz que forma o cenário.

A partir de uma posição é possível obter todas aquelas vizinhas à atual. Com esta divisão podemos utilizar diversos algoritmos para busca de caminhos entre as diversas posições alcançáveis dentro do mundo.

Ilustração 8 – The Legend of Zelda: Alink of the Past – Exemplo de jogo 2D

Fonte: Nintendo Wii Games, Wikis, Cheats, News, Reviews & Videos[3]

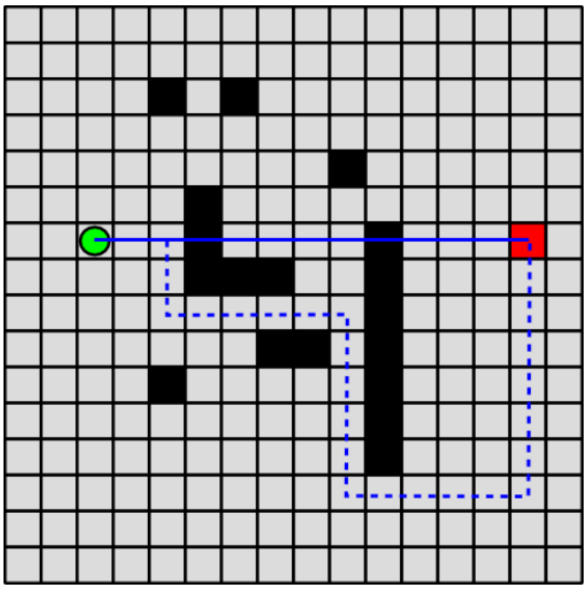
Este trabalho utilizará a última opção para desenvolvimento dos algoritmos, uma vez que a implementação do método *Tile* pode ser considerada uma extensão da divisão em *grid.*

A partir da representação do mundo, pode-se desenvolver diversos algoritmos que serão responsáveis pela movimentação dos personagens. Para que um agente possa se deslocar de um ponto a outro, é necessário que o algoritmo seja capaz de "reconhecer" as partes que formam o mundo onde o agente está inserido. Ou seja, ele deve ser capaz de traçar um caminho até seu objetivo, mas levando em consideração as partes alcançáveis e aquelas

[3] Fonte: Disponível em <http://wiiign.com>

onde não pode passar (paredes, água, etc). A Ilustração 9 apresenta um caminho encontrado por um algoritmo de *pathfinding*, ou de busca de caminho em um dado *grid*. Pode-se perceber a linha tracejada, como o caminho a ser seguido pelo agente e a linha azul como o caminho mais curto até o objetivo, se não houvesse obstáculos.

Ilustração 9 – Busca de caminho



Fonte: Exemplificação do método

A otimalidade de um algoritmo de busca de caminho diz respeito à solução que apresenta menor custo em relação a todos os caminhos possíveis. Essa otimalidade é relativa à que tipo de custo é preferível ao jogo em questão. Por exemplo, em um jogo o caminho ótimo pode ser aquele mais curto, ou seja, aquele que o agente "anda menos"; mas em outro, o caminho ótimo pode ser aquele em que o agente encontra menos inimigos e, assim gasta menos munição ao executá-los.

No desenvolvimento destes algoritmos devem ser observadas diversas características do jogo em questão, como por exemplo: "qual caminho é o melhor: o mais curto ou o que foi encontrado primeiro?", "O mundo pode mudar antes que o personagem chegue ao objetivo?".

As questões acima, aliadas ao que se espera do agente dentro do jogo (um agente não precisa achar um caminho ótimo na maioria das vezes, por exemplo), são responsáveis pela decisão de que algoritmo pode ser melhor aproveitado em determinado jogo. Entretanto, devemos considerar também o uso de recursos no processo de escolha do melhor algoritmo de busca de caminho.

O uso de recursos de um algoritmo de busca de caminho deve ser levado em conta pois, o desempenho do *hardware* está diretamente ligado ao processo de ambientação do jogador ao jogo. Muitas vezes, quando o espaço de busca (ambiente) é muito complexo, ou seja, há muita informação a ser guardada, o uso de memória por parte dos algoritmos de busca de caminho se torna elevado. Um exemplo é o algoritmo de busca em largura que, se o ambiente é muito grande, precisa guardar muitos nós em memória para armazenamento do caminho já percorrido.

Um grande uso de memória pode causar efeitos indesejados no momento de execução do jogo. Por exemplo, o jogo pode "trancar" cada vez que um agente precisa buscar seu caminho se o algoritmo de navegação usa muita memória do aparelho onde o jogo está sendo executado.

Para que isto não ocorra é necessário escolher o melhor algoritmo para cada plataforma, o que acaba aumentando a quantidade de versões que devem ser desenvolvidas para um jogo e, consequentemente aumentando o custo total de produção do mesmo. Na maioria das vezes é preferível inserir melhorias para que a execução do algoritmo possa ser melhorada, mesmo que a otimalidade possa ser prejudicada.

No presente trabalho, pretende-se testar diversos algoritmos utilizando a linguagem *Action Script 2* (AS2) e a plataforma Flash, atualmente muito utilizadas em jogos para celulares, *tablets* e *browsers*. Levando-se em conta as usuais configurações de hardware destas tecnologias, almeja-se propor melhorias, quando possível, quanto à economia de memória na implementação destes algoritmos.

2.10. Estado da Arte

Geralmente para cada tipo de jogo são implementadas técnicas de inteligência artificial que melhor se encaixem no propósito do jogo. Por exemplo, as técnicas utilizadas em um jogo de xadrez, poderiam não ser aplicáveis em um jogo de tiro em primeira pessoa. O que torna a área da inteligência artificial em jogos muito ampla, pois se tomarmos 100 jogos como exemplo, teremos, provavelmente, 100 formas diferentes de implementação de inteligência. Deve-se ter em mente que não há padrão de lógica que possa ser aplicado a todos os personagens do jogo, o que gera maiores modificações nos algoritmos, uma vez que devem ser implementados de maneiras diferentes para que o efeito de realidade seja alcançado.

Os jogos de tiro em primeira pessoa, como *Counter-Strike* (Ilustração 10), geralmente não exigem uma implementação muito forte de inteligência artificial, já que sua jogabilidade consiste principalmente em movimentar os personagem principal e atirar nos oponentes. A técnica de inteligência artificial mais comum utilizada neste tipo de jogo é a FSM (Máquina de Estados Finita), onde há um conjunto de estados e regras que definem as transições entre estes estados. Por exemplo, se o NPC está no estado "Vivo e Atirando" e é atingido, deve utilizar das regras para mudar de estado para "Morto" ou para "Esquivar e Atirar". O ponto negativo desta técnica é a previsibilidade, ou a extensão de regras e estados quando o ambiente é mais complexo. Outra técnica utilizada é a lógica Fuzzy, capaz de representar melhor eventos não contáveis ou imprecisos, como "quantidade de felicidade" ou "pouca comida". Segundo Karlsson (2005), há jogos que misturam a lógica Fuzzy nos estados de transição das FSMs, obtendo uma aparência de inteligência mais refinada, como no caso do jogo *Unreal*.

Ilustração 10 – Jogo de tiro em primeira pessoa, *Counter Strike*

Fonte: Blog 2BP[4]

Os jogos de estratégia, que exigem raciocínio lógico do jogador (para evoluir uma simples vila até um império, controlar tropas e armamentos, etc.), como *Command & Conquer* (Ilustração 11), necessitam de técnicas que sejam capazes de gerenciar a navegação

[4] Disponível em <http://2.bp.blogspot.com/-3QbFtiTjVKE/TrvuElKgb9I/AAAAAAAAChU/EZCwlraWAXk/s1600/Counter+Strike+1.6+Screen+Shot+2.jpg>

de unidades e aplicar planejamentos estratégicos próximos à realidade (ritmo de produção, por exemplo). Geralmente é utilizada algoritmos como o A* para as movimentações dos NPCs e máquinas de estado aliadas a sistemas baseados em regras parametrizáveis (testes de condição comparadas com as regras pré-estabelecidas para a situação atual) para ajuste de dificuldade, jogabilidade e evolução. (FLAUSINO, 2007). Outra técnica (VIEIRA, 2005) é a criação de "planos de ação", os quais podem ser escolhidos pelo personagem, o que o torna menos previsível.

Ilustração 11 – *Command & Conquer,* jogo de estratégia

Fonte: Blog Game It All[5]

Os jogos de esporte, onde os que mais se destacam são os jogos de futebol como o *Fifa Soccer* (Ilustração 12), precisam de um nível mais refinado de programação de inteligência, uma vez que devem reproduzir o comportamento humano. Neste caso, a inteligência artificial não pode ser perfeita em suas ações. Por exemplo, jogadores reais erram passes, se machucam, caem, cansam e, se a inteligência artificial empregada nos NPCs torná-los "perfeitos" a imersão do jogador será prejudicada, pois a dinâmica do jogo destoará da realidade esperada. Para isso, muitas vezes são utilizadas redes neurais, técnica que simula o cérebro humano tendo nós como neurônios e conexões entre esses nós que guiam a uma saída (pensamento), armazenando conhecimento gerando aprendizado. Podemos citar a evolução de um time numa partida: se o jogador está enfrentando uma equipe pela primeira vez, espera-se

[5] Disponível <http://gameitall.com/wp-content/uploads/2012/08/ccuc.jpg>

que no início os jogadores adversários não consigam interceptar todos os passes, pois “ainda estão absorvendo o jeito do outro time jogar”, entretanto com o passar do tempo eles podem “aprender” qual a melhor maneira de marcar cada jogador controlado pelo usuário. Alguns jogos utilizam a lógica Fuzzy (VIEIRA, 2005) para situações que não podem ser representadas satisfatoriamente por um número, como o quão forte foi uma falta.

Ilustração 12 – Fifa Soccer, jogo de esporte

Fonte: http://s2.n4g.com/media/11/newssi/55000/57674_0_org.jpg

Muitas são as técnicas utilizadas nos diversos tipos de jogos e tudo se torna mais complexo a medida que estas técnicas precisam ser combinadas. E, como citado por TATAI (2003), as soluções computacionais desenvolvidas pela indústria de jogos tradicionalmente são mantidas em sigilo, o que torna complicado definir exatamente quais técnicas são usadas em cada jogo.

2.11. Trabalhos Relacionados

Há vários trabalhos que tratam da aplicação da inteligência artificial na área de jogos, entretanto nenhum foi encontrado que trate do estudo de diferentes tipos de algoritmos aplicados em ambientes moldados usando o método *Tile* e levando em conta as limitações de hardware da maioria dos aparelhos onde este método é empregado.

Alguns trabalhos, como de Tatai (2006), Augusto (2009) e Aline Silva (2009) tratam de um estudo da inteligência artificial e sua aplicabilidade nos jogos, explicando os

conhecimentos básicos, mas não se atendo a aplicação direta ou na comparação das possibilidades desta.

No artigo desenvolvido por Bonfandini (2011), podemos ter uma visão de diversas técnicas usadas na aplicação de busca de caminho, que explica as lógicas envolvidas e os algoritmos mais conhecidos nesta área. Nele é apresentada ainda uma comparação entre estes algoritmos, quanto a uso da memória, processamento e otimalidade. Entretanto, não leva em consideração os recursos da máquina onde serão executadas as buscas.

Na monografia desenvolvida e apresentada por Paulo Roberto Lafetá Ferreira (2009), é apresentado o uso de busca de caminho em jogos, utilizando *Navigation Meshes* como um grafo de navegação, para aplicar o algoritmo A* e a técnica *Potential Function-Based Movement* (movimento baseado em função potencial) para o desvio de obstáculos dinâmicos. Este trabalho foi capaz de mesclar o algoritmo A* com o uso de forças potenciais, resultando em um movimento mais sutil e próximo ao natural ao fazer o NPC desviar do objeto dinâmico e em seguida retornar ao caminho pré-determinado. Já o relatório apresentado por Pastor e Corrêa (2010) aborda duas soluções para navegação em mundos virtuais: a primeira baseada em Inteligência Artificial usando o algoritmo A*; a segunda baseada na teoria de atratores (Steering Behavior). O artigo ainda propõe uma abordagem híbrida utilizando as duas abordagens, obtendo um sistema complexo e que necessita de um grande processamento, mas que apresenta bons resultados para plataformas onde a limitação do hardware não precisa ser levada em consideração.

O site mantido por Tonypa (2003 – 2005) apresenta uma visão bem ampla sobre o desenvolvimento de *Flash Tile Games*. Neste site, que é liberado sob Creative Common License, é abordada a busca de caminho neste tipo de jogo, mas apenas os algoritmos Breadth-First Search e Best-First Search, sem no entanto apresentar uma comparação detalhada contundente entre elas.

Michael Grundvig (2004) desenvolveu uma variante do algoritmo de *Djikstra* sem suposições de heurísticas aplicável a *Flash Tile Games*. Este diminuiu bastante o tempo que um agente demora de um ponto A até um ponto B incluindo uma pré-programação de busca de caminho. Entretanto, ele cita que o tempo de pré-processamento é elevado.

Liu *et al* (2010) aborda o uso da inteligência artificial na aprendizagem móvel, uma nova forma de aprender usando o celular e a tecnologia móvel para acesso à educação, informação, recursos educacionais e serviços de educação. Neste contexto, a inteligência artificial é usada para selecionar o conteúdo de aprendizagem adequada e seu ajuste ao

progresso do aprendizagem do aluno, assim como na avaliação da curva de aprendizagem, análise dos erros e atitudes corretivas para sanar dúvidas.

Xu e Doren (2011) propõe um sistema baseado em busca de caminho para visitação de museus. O *Museum Visitors Guide* é um sistema que usa uma modificação do algoritmo A*. A área do museu é representada por um mapa de tiles onde o visitante pode marcar as obras que quer ver e o sistema calcula dinamicamente o caminho até elas baseado no local onde está no momento. O sistema deve ser executado em computadores distribuídos em áreas chaves do museu. Citam que estão produzindo uma versão deste software para dispositivos que utilizam Android.

Aiolli e Palazzi (2008) asseguram que uma boa inteligência artificial é crucial para tornar um jogo desafiador. Consideram também, a complexidade em desenvolver módulos de inteligência artificial em dispositivos móveis, pelo seu *hardware* limitado ou até mesmo em computadores mais poderosos, mas considerando jogos nos quais não se pode ter uma visão completa do estado do jogo a qualquer momento (*Poker,* por exemplo). Propõe melhorar estes problemas utilizando um algoritmo de aprendizagem de máquina que procura inferir a informação que está em falta. Desenvolveram um jogo (*Ghosts)* utilizando Java ME para dispositivos móveis que utiliza esta técnica de aprendizagem de máquina baseado em protótipos. Os resultados mostram claro ganho de desempenho preditivo, principalmente em dispositivos com recursos limitados, onde um algoritmo clássico de busca pode não ser o indicado.

Koller (2007), comparam as plataformas Flash Lite e Java ME para jogos estilo *arcade* para dispositivos móveis. O jogo foi desenvolvido utilizando uma malha de *tiles*, usando matrizes bidimensionais para armazená-la e a notação de que cada *tile* é um objeto com características próprias. Foram avaliadas diversas características do jogo obtido, dentre elas: espaço do arquivo final, uso de memória e *frame rate*.

Xin (2009) trata da aplicação de inteligência artificial em *Serious Games* executados em celulares. Analisa as possíveis falhas ao executar IA em dispositivos móveis, assim como propõe melhorias. Discorre que, ao contrário dos jogos comuns, o principal objetivo dos *Serious games* não é o entretenimento por si só, geralmente são usados para educar e/ou treinar. Afirma que, pelo baixo poder de processamento dos celulares e também sua pouca memória, é indicado o uso de técnicas básicas de IA, geralmente baseadas em regras determinísticas, fluxo e procedimentos. Entretanto, para *Serious Games* esse tipo de técnica não é suficiente. Recomenda a otimização dos algoritmos de IA, assim como o uso, se

possível, das conexões disponíveis no aparelho, usando um servidor remoto para rodar o algoritmo e em seguida reenviar ao aparelho do usuário.

Duh *et al* (2010) faz um comparativo sobre as experiência dos usuários jogando em dispositivos móveis (telefones). Segundo seu estudo, ficou explícito que alguns jogos são melhores de ser jogados em celulares que outros e que, portanto, é melhor que os designer façam seus jogos voltados para cada tipo específico de dispositivo. Para a avaliação foram consideradas algumas características dos jogos como, facilidade de controle e níveis de dificuldade.

3 RECURSOS, FERRAMENTAS E MÉTODOS

Para a realização do presente trabalho foi desenvolvida uma série de protótipos de aplicação com ambiente baseado em *Tiles,* onde serão testados alguns algoritmos de busca de caminho. O programa *Adobe Flash* foi utilizado para o desenvolvimento e a linguagem de *script Action Script 2* para a programação do mesmo. O protótipo foi baseado nos códigos disponibilizados por Tonypa[6] (2003 – 2005) sob a licença *Creative Commons License* e modificado segundo as necessidades para a correta aplicação das técnicas abordadas neste trabalho, uma vez que os códigos de Tonypa são apenas exemplos de aplicação e devem ser combinados e/ou modificados para que um jogo completo possa ser produzido.

Foram produzidos dois tipos de protótipo para cada algoritmo de busca de caminho a ser testado. Estes se caracterizam como um jogo completo no estilo defesa de território, que será melhor detalhado no capítulo a seguir.

Todos os protótipos possuem seu ambiente montado em formato de matriz, de acordo com o método *Tile.* Cada *Tile* será gerenciado como um objeto (para que possam ser atribuídas suas propriedades, como *sólido, por exemplo*). A matriz que armazena o ambiente será composta por números que indicam qual o objeto que ocupará qual lugar no momento de montar o cenário.

Veja o exemplo abaixo:

```
mapa1 = [
        [1, 1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 0, 1],
        [1, 0, 1, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 1, 0, 1],
        [1, 0, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 1, 1, 1]
    ];
```

Neste exemplo, o ambiente seria composto por 6 linhas e 8 colunas e apenas dois tipos de *Tiles* (0 e 1).

O que diferenciará os protótipos serão os tipos de algoritmos utilizados e o tamanho do *grid* que representa o plano de fundo do mesmo. O sistema será definido inicialmente como: completamente observável, determinístico, sequencial, estático e discreto. Ou seja, o agente pode visualizar todo o mundo e o estado deste pode ser previsto completamente. É sequencial,

[6] Disponível em http://www.tonypa.pri.ee/tbw/

pois para achar o caminho o agente depende de todas as ações desenvolvidas até o momento. É estático, pois o mundo não muda enquanto o agente calcula qual ação usar e discreto, pois há um número finito de ações, estados e percepções.

Os testes foram desenvolvidos em cima das ferramentas disponibilizadas para testes de aplicativos em plataformas móveis (*Adobe Device Central*), onde é possível visualizar a execução do aplicativo em diversas plataformas móveis e apresenta informações como uso de memória em tempo de execução e tamanho da aplicação.

Game Design

Pode-se definir *Game Design* como um processo global que envolve a análise, especificação e criação das regras e características de um jogo a ser desenvolvido. Envolve o planejamento das regras, dinâmicas e elementos de um jogo. É onde é definido desde a jogabilidade até as consequências das ações do jogador dentro do jogo. A seguir será apresentado na forma de uma análise simples de *Game Design*, o jogo desenvolvido para a realização do presente trabalho.

3.1. Nome do Jogo

Territory (território, em português). Na Ilustração 13, pode-se observar a tela inicial apresentada ao abrir o jogo.

Ilustração 13 – Tela Inicial

Fonte: Execução do protótipo do jogo (Imagem baseada em diversas imagens de barbaros encontradas na internet)

3.2. Visão Geral do Jogo

3.2.1. Objetivos de desenvolvimento

Territory é um jogo desenvolvido como parte integrante e fonte de recursos de pesquisa para este trabalho. Através dele, pretende-se testar diferentes tipos de algoritmos de busca de caminho e verificar o comportamento do mesmo em dispositivos móveis.

3.2.2. História

O personagem principal é Jötnar (Ilustração 14), um bárbaro que tem um esconderijo de riquezas em terras à noroeste da Ásia (subentende-se como um continente de um planeta não citado na estória).

Ilustração 14 – *Sprite* personagem principal, Jötnar

Fonte: http://tsgk.captainn.net/?p=showgame&t=sy&sy=10&ga=377

Entretanto, Jötnar passa a ter problemas quando 3 espécies de pássaros-do-fogo, pássaros velozes e furiosos cujo habitat é o mesmo onde o bárbaro resolveu guardar seus saques, começam a saqueá-lo, para que possam construir seus ninhos com o ouro e as pedras preciosas acumuladas pelo heroi.

Depois de algum tempo sendo saqueado, Jötnar decide utilizar seus conhecimentos em magia adquiridos em suas viagens pelo mundo para afastar esses perigosos animais do seu tesouro.

3.2.3. Jogabilidade

O jogador controla Jötnar e deve defender seu território. Para isto ele deve distribuir poções pelo caminho que são capazes de destruir os pássaros para que estes não roubem seu tesouro ou que, pelo menos, consigam carregar uma menor quantidade de ouro. Ao mesmo

tempo, o bárbaro deve estar atento, pois podem aparecer itens em seu território que podem ajudá-lo em sua missão de defesa.

3.2.4. Controles

As setas direcionais e números (2,4,6,8) movimentam o personagem e o número 5 coloca uma poção no local onde o bárbaro está no momento ou a recolhe caso já haja uma poção na posição onde o bárbaro está.

3.2.5. Características do jogo

O objetivo do jogo é não deixar que nenhum dos três baús de tesouro do bárbaro fique vazio;

Cada pássaro tem uma característica específica:

- Pássaro-do-fogo branco (Ilustração 15) – É o mais fraco, morre com apenas uma poção. Tem maior probabilidade de aparecer. Pega $100 ouros do baú para o qual se dirige.

Ilustração 15 – *Sprite* pássaro-do-fogo branco, mais brando

Fonte: http://tsgk.captainn.net/?p=showgame&t=sy&sy=10&ga=377

- Pássaro-do-fogo amarelo (Ilustração 16) – Inimigo de dificuldade média. Precisa passar por duas poções para morrer. Tem menos probabilidade de aparecer que o pássaro-do-fogo branco. Pega $200 ouros do baú o qual saqueia.

Ilustração 16 – *Sprite* pássaro-do-fogo amarelo, levemente irritadiço

Fonte: http://tsgk.captainn.net/?p=showgame&t=sy&sy=10&ga=377

- Pássaro-do-fogo cinza (Ilustração 17)– É o mais feroz. Só pode ser derrotado com três poções, entretanto é o que tem menor probabilidade de aparecer. Pega $400 ouros em cada ataque.

Ilustração 17 – *Sprite* pássaro-do-fogo cinza, o furioso

Fonte: http://tsgk.captainn.net/?p=showgame&t=sy&sy=10&ga=377

Os inimigos não possuem posição inicial fixa. Podem aparecer em qualquer *tile* da linha superior do jogo. Cada inimigo pode se dirigir para um baú diferente.

O jogador e os pássaros não podem passar pelos tiles que representam água, ou seja, devem contorná-los.

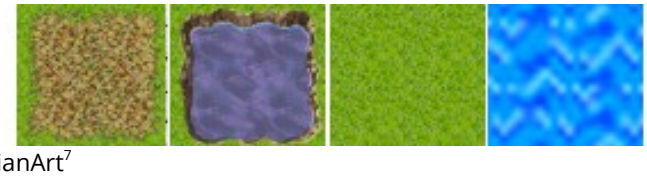
O jogador não morre ou perde vida se encostar nos pássaros.

3.3. Escopo

3.3.1. Locais

O jogo possui um único local, o território do bárbaro. Que é montado baseado nos *tiles* representados na Ilustração 18.

Ilustração 18 – *Sprite* contendo os tiles utilizados para montar o jogo



Fonte: DevianArt[7]

3.3.2. Fases

A versão do jogo para o TCC possui apenas uma fase.

3.3.3. NPCs

Possui 3 NPCs. Os pássaros-do-fogo, que podem ser vistos nas Ilustrações 15,16 e 17.

3.3.4. Armas

Apenas o personagem principal possui "armas", as poções (Ilustração 19). Cada pássaro precisa de um determinado número de poções para ser derrotado, como pode ser visto na seção 3.2.5 deste.

---

[7] Fonte: http://fc08.deviantart.net/fs70/f/2010/263/0/e/mack__s_tile_set__a2_by_draymondc-d2z3z4d.png e http://pousse.rapiere.free.fr/tome/

Ilustração 19 – Item poção: são distribuídas pelo jogador no território e tem a capacidade de retardar ou destruir pássaros-do-fogo

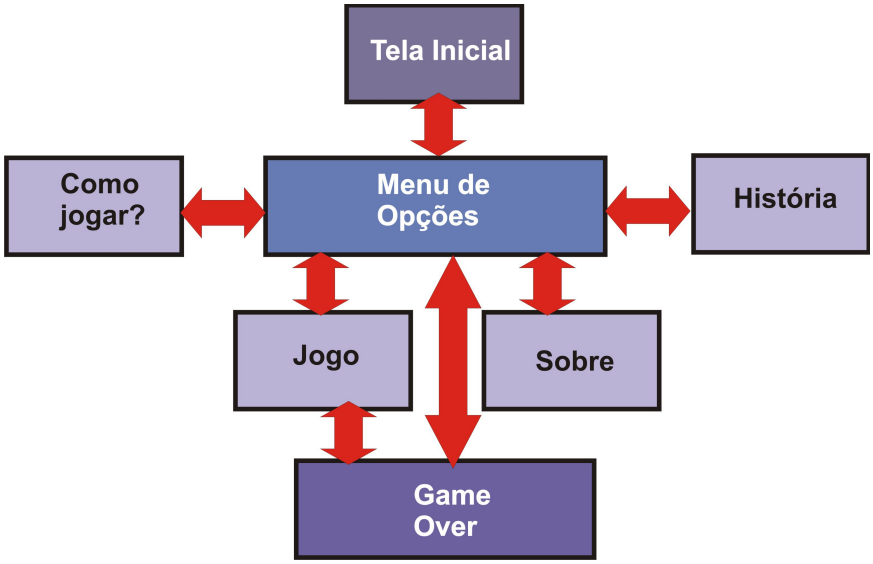
Fonte: TomeTik[8]

3.4. Mecânica

3.4.1. Fluxo de Telas

A Ilustração 20 apresenta as telas do jogo Territory e as ligações entre elas, demonstrando o fluxo que pode ser obtido ao executar a aplicação.

Ilustração 20 – Fluxo de telas do jogo



3.4.2. Tela Inicial

Na primeira tela (Ilustração 13) há uma imagem estilizada de um bárbaro, que representa o personagem principal – Jötnar e um botão que, ao ser pressionado levará o jogador ao menu de opções do jogo.

[8] Fonte: http://pousse.rapiere.free.fr/tome/

3.4.3. Menu de Opções

Na 2º tela, o usuário pode escolher entre 4 opções:

- Jogar – Começa o jogo imediatamente.
- História – Apresenta a história do jogo. Responsável pela imersão inicial ao jogo, onde o usuário compreende o universo do aplicativo e o que deve fazer.
- Como Jogar – Mostra quais teclas são utilizadas no jogo e suas funções
- Sobre – Detalhes sobre programador, design e objetivo geral do protótipo.

3.4.4. O jogo

Para o desenvolvimento deste projeto, Territory tem duas variantes do protótipo padrão. A diferença entre essas duas versões está no tamanho dos *tiles* que compõem a imagem de fundo. No protótipo1 (Ilustração 21) os *tiles* tem o tamanho 20x20 pixels e o campo de jogo tem as dimensões 12x15 *tiles*. No protótipo2 (Ilustração 22), os *tiles* tem o tamanho 10x10 pixels e o campo de jogo tem as dimensões 24x29 *tiles*.

Ilustração 21 – Protótipo1 – campo de jogo: 12x15px

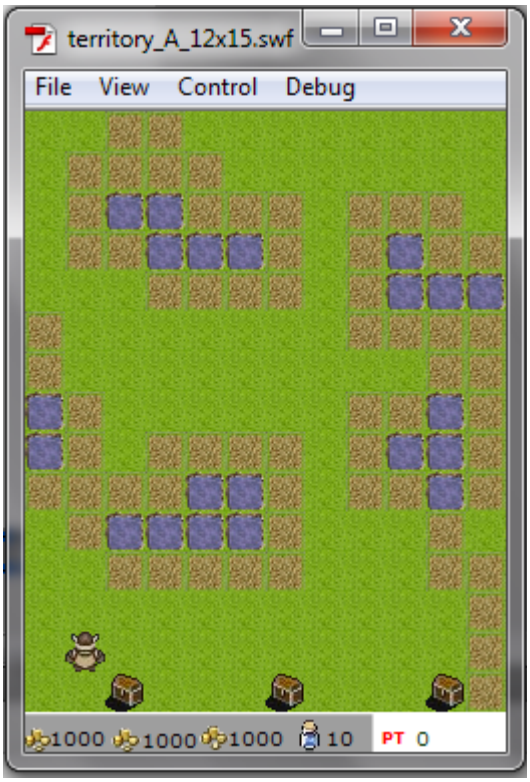
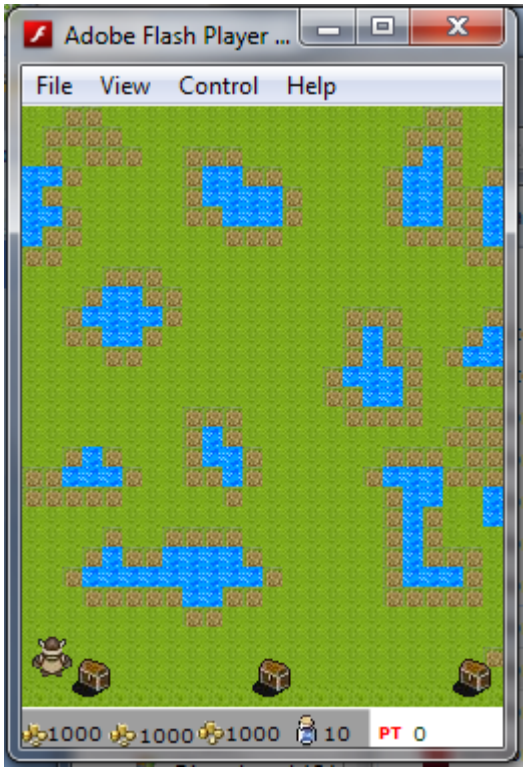


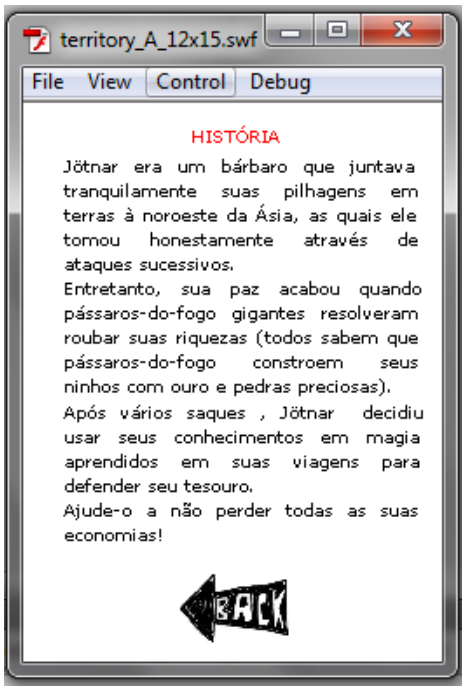
Ilustração 22 – Protótipo2 – campo de jogo: 24x29px



3.4.5. História

Nesta tela (Ilustração 23), o jogador pode conhecer a história que norteia o jogo. Mais uma vez busca-se a imersão do jogador e a identificação do mesmo com o personagem principal.

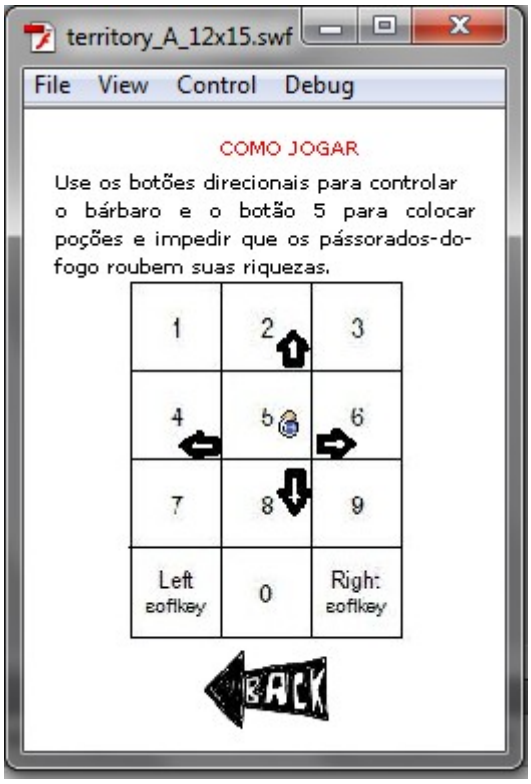
Ilustração 23 – Protótipo: História



3.4.6. Como jogar

Esta tela visa ensinar ao jogador quais botões deve usar para controlar o jogo e, mais uma vez, reforçar o objetivo do jogo. Ver Ilustração 24.

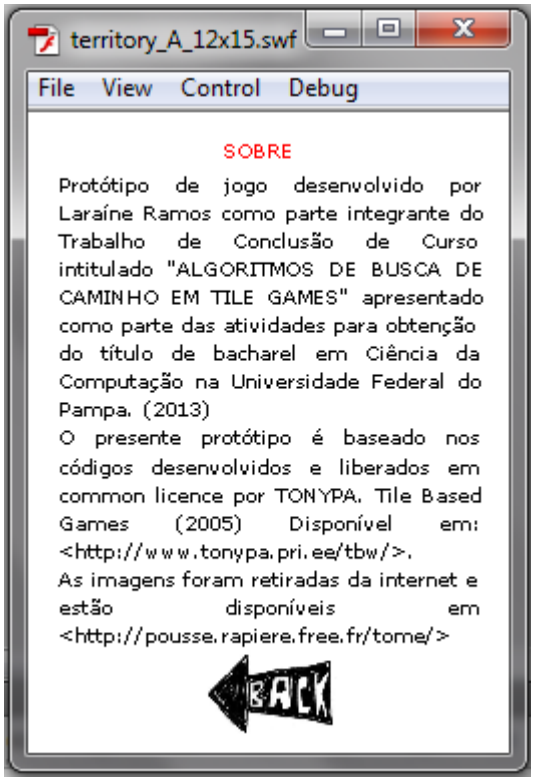
Ilustração 24 – Protótipo: Como jogar



3.4.7. Sobre

Contextualiza o ambiente de desenvolvimento do jogo, apresentando o título da monografia para a qual foi desenvolvida e as referências de onde o código base foi retirado. A Ilustração 25 mostra o *screenshot* desta tela.

Ilustração 25 – Protótipo: Sobre



3.4.8. Funcionamento Geral

O protótipo padrão foi desenvolvido para telas de 240x320 pixels de resolução, entretanto, se ajusta caso a tela do dispositivo seja maior. O cenário do jogo foi desenvolvido baseado no método *tile,* e é representado por uma matriz, onde cada posição representa um *tile.* Assim, o posicionamento de todos os objetos em cena é declarado em forma de coordenadas *(x,y)*.

O jogo foi modelado para ocupar toda a tela do dispositivo e não mostrar menus próprios do sistema operacional do aparelho.

Em sua programação, há um objeto geral *Game*, o qual é responsável por guardar todas as características do jogo, tais como: tamanho do *tile* usado no jogo, mapa corrente, número de inimigos vivos no momento, tempo, profundidade dos inimigos e profundidade dos itens em cena (a profundidade dos objetos é utilizada para que um objeto não sobreponha os outros, é formada uma pilha de profundidades onde a imagem de menor profundidade é sobreposta por todas as outras e de maior valor não é sobreposta por nenhuma outra imagem).

Em seguida são determinadas as propriedades para os tipos de *tiles*, que incluem os atributos: nome, *walkable* (se os personagens podem andar pelo *tile* ou precisam contorná-lo) e o gráfico que o representa.

Logo depois são declarados os ítens que podem ser usados em cena, os quais possuem atributos que definem seus nomes, se o personagem pode passar por cima deles, a imagem e a sua probabilidade de aparecer em jogo

É definido então um vetor que armazena as posições iniciais dos inimigos. Cada vez que um inimigo é criado, uma posição inicial é gerada randomicamente para ele.

Outro vetor armazena as posições dos potes de ouro, ou seja, objetivo final dos inimigos.

Um terceiro vetor armazena as posições onde os itens podem aparecer no jogo, quanto mais posições, mais dificíl de o jogador prever onde podem aparecer os itens seguintes e mais ele tem de se afastar dos baús de tesouro, o que representa uma maior dificuldade de proteção do território.

A seguir, são definidas as propriedadees que definem os inimigos: vida, quantidade de ouro retirada a cada ataque, velocidade, nome, probabilidade de aparecer, se está se movendo, posição *x* inicial, posição *y* inicial, e o caminho que percorrerá até chegar ao seu destino. O caminho é calculado no momento que o inimigo entra em campo.

O personagem principal é um objeto que possui as propriedades: posição inicial *x*, posição inicial *y*, velocidade, se está se movendo, quantidade de ouro no baú1, quantidade de ouro no baú2, quantidade de ouro no baú3, energia e pontos.

Assim, que a tela de jogo é aberta, a função *CriaMapa(map)* é chamada. Ela recebe uma matriz representando o terreno e as posições que cada *tile* ocupará no cenário. Em seguida, ela percorre a matriz montando o plano de fundo do jogo. Os baús de ouro são colocados em suas posições em um nível (profundidade) acima dos *tiles* do terreno. O personagem principal é então adicionado.

O jogo é mantido em um *loop* no *frame* do jogo, onde detecta-se o pressionamento de teclas e a movimentação dos inimigos. Conforme a tecla pressionada, é chamada a função de movimentação do personagem ou a função responsável por deixar uma armadilha no local.

A função *moveChar(ob, dirx, diry)* recebe o objeto que será movimentado e a direção em *x* e *y* que ele deve se mover. A função verifica se o *tile* para onde o personagem quer se mover é andável ou não. Ser for, modifica a posição do personagem, senão apenas o aproxima do obstáculo, o qual ele não pode ultrapassar. Em seguida verifica se há algum item no tile

que o personagem passou a ocupar, se sim aplica o bônus cabível ao tipo de item e o remove da posição, identificando que o persongam já o pegou, ou seja, ele já não está mais disponível em cena.

Há um contador que a cada 3 segundos, decide se deve colocar um novo item em cena ou um inimigo. Se decidir colocar um item, a função *colocaItems()* é chamada. Nesta função um item é sorteado segundo suas probabilides e colocado em uma das posições possíveis dentro do jogo. Se a escolha for por um novo inimigo a função *colocaInimigo(qual)* é chamada, na qual um tipo de inimigo é sorteado segundo suas probabilidades e é inserido aleatóriamente em um dos *tiles* iniciais possíveis. O novo inimigo é colocado na lista de inimigos ativos, é definido qual dos baús é seu objetivo e um caminho até ele é encontrado e armazenado.

Continuamente a função *moveEnemies* percorre a lista de inimigos existentes movimentandos-os no caminho pré-definido até seu objetivo e verificando a cada passo se não há uma poção agindo sobre o personagem. Se houver, calcula se aquela poção acaba com toda a vida do inimigo, se sim o retira do jogo e da lista de inimigos, caso contrário, diminui sua quantidade de energia e o movimenta mesmo assim. Também é verificado se o inimigo encontrou um baú, se sim, o valor definido para o tipo de inimigo é retirado do baú e o inimigo é excluído da cena e da lista de inimigos atuantes.

3.5. Física

O jogo foi desenvolvido com base em duas dimensões e funciona em 12 FPS (frames por segundo).

Todas as entidades destrutíveis, assim como o personagem possuem um determinado valor que representa os seus pontos de vida.

Os pontos de vida do jogador são representados pela quantidade de ouro existentes nos baús que ele está protegendo.

Os inimigos tem seus pontos de vida decrescidos em 1 para cada poção em que encostam e também, neste caso a quantidade de ouro que podem carregar decresce em $50.

Um inimigo é destruído quando seus pontos de vida chegam em 0.

O personagem principal é destruído quando a quantidade de ouro em um de seus baús chega a 0.

Em qualquer momento do jogo, o jogador pode recolher uma poção e recolocá-la em qualquer outro lugar do cenário.

As quantidades de vida e de ouro retirados pelos inimigos são:

- Pássaro-do-fogo branco: 1 vida e retira 100 ouros
- Pássaro-do-fogo amarelo: 2 vidas e retira 200 ouros
- Pássaro-do-fogo cinza: 3 vidas e retira 400 ouros

Para coletar um item, o jogador precisa ocupar o mesmo *tile* que o ítem pretendido, ou seja, é preciso "passar" por cima do ítem para pegá-lo.

Os pássaros retiram ouro do baú assim que chegam no mesmo *tile* do baú, completando assim sua missão e sendo retirados do cenário e da lista de inimigos ativos.

Não há limitação de quantidade de inimigos em jogo.

3.5.1. Combate

O jogador deverá colocar as poções no caminho dos pássaros-de-fogo para destruí-los antes que estes possam roubar o seu ouro. As Ilustrações 26 e 27 mostram *scrensHots* da execução de dois protótipos.

Ilustração 26 - Protótipo1: Exemplo de execução do jogo

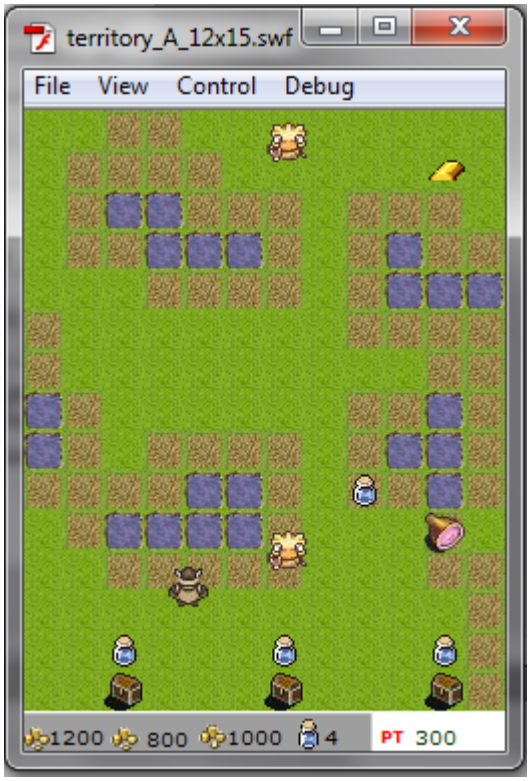
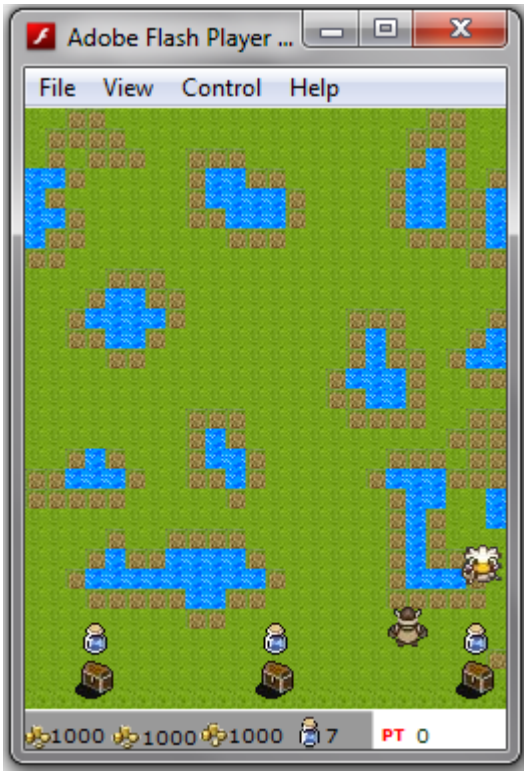


Ilustração 27 – Protótipo 2: Exemplo de execução do jogo



3.5.2. Economia

O jogador recolherá itens durante o jogo. Estes itens podem ser:

- Barra de ouro – acrescenta 200 ouros em um dos baús de Jötnar, o qual é sorteado randomicamente. Tem 40% de probabilidade de aparecer. Ver Ilustração 28.

Ilustração 28 - Item barra de ouro: acrescenta 200 ouros a um dos baús do bárbaro

Fonte: http://pousse.rapiere.free.fr/tome/

- Coração – acrescenta 100 ouros em todos os baús do bárbaro. Tem 10% de probabilidade de aparecer. Ver Ilustração 29.

Ilustração 29 - Item coração: acrescenta 100 ouros a cada um dos baús de Jötnar

Fonte: ttp://tsgk.captainn.net/dld.php?s=snes&f=bszelda_items_sheet.png

- Pernil – Permite que o bárbaro coloque duas poções a mais em campo. Tem probabilidade de 50% de aparecer. Ver Ilustração 30.

Ilustração 30 - Item comida: faz com que o bárbaro possa produzir mais duas poções

Fonte: http://pousse.rapiere.free.fr/tome/

3.5.3. Pausando, começando o jogo novamente e mecanismos para salvar o progresso

Se as vidas do jogador acabarem, ele deve recomeçar do início. Não foi implementado nenhum artifício para salvar o progresso do jogo. Não é possível colocar o jogo em pausa.

3.6. Algoritmos de busca de caminho

Como citado anteriormente, os protótipos desenvolvidos durante a realização do presente trabalho testarão alguns algoritmos de busca de caminho, que são aqueles responsáveis por prover um caminho entre dois pontos quaisquer em um determinado plano.

Os algoritmos de busca de caminho costumam se dividir em duas categorias: busca cega e busca informada.

Algoritmos de **busca cega** não utilizam nenhuma informação sobre o objetivo final. Não se importam com os custos de achar uma solução, apenas testam nós vizinhos até que alguma solução seja encontrada. Deste modo, é fácil perceber que nem sempre é ótima.

Aliás, sua definição de melhor solução, é aquela que está mais perto da raíz da árvore de busca (RIBEIRO, 2013).

A maioria dos algoritmos utiliza uma grande quantidade de memória, pois expande a maioria dos nós do espaço de busca, uma vez que não classifica sua proximidade ao nó objetivo.

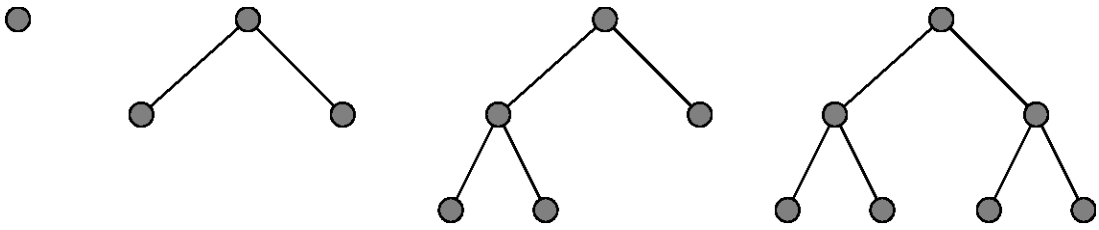
Na **busca informada**, é introduzido o conceito de custo. Esse custo é definido com base na aplicação de uma função que permite classificar os nós. Quanto menor o valor retornado por esta função, mais chances o nó tem de estar no caminho para a solução. Para os algoritmos desta categoria, o custo da solução desempenha o papel principal na aceitação da mesma. Geralmente, isto gera uma certa economia de memória, uma vez que alguns nós não são expandidos por não se encaixarem na definição do problema.

A seguir são apresentados os algoritmos que foram implementados para o desenvolvimento deste trabalho, tomando o algoritmo de Busca em Largura como algoritmo base de comparação para com os outros.

3.6.1. Busca em Largura (B*readth-First Search* = *BFS*)

A busca em largura está intimamente relacionada com os conceitos de distância e caminho mínimo, fazendo uma varredura por "níveis" (Ilustração 31[9]). Isto é, começando pelo vértice incial *S*, o algoritmo visita todos os vértices que estão a uma distância 1 de *S*, em seguida visita todos os vértices que estão à distância 2, e assim por diante. (FEOFIOLOFF apud SEDGEWICK, 2002)

Ilustração 31 - Exemplo de execução do algoritmo de busca em largura



Fonte: http://www.gsigma.ufsc.br/~popov/aulas/ia/modulo3/index.html

Seu algoritmo é classificado como parte da categoria de **busca cega**.

É **completa**, ou seja se o nó objetivo estiver estiver em uma profundidade *d*, o algoritmo o encontrará assim que tiver expandido todos os nós mais rasos. **Ótima**, garante que o caminho mais raso foi encontrado.

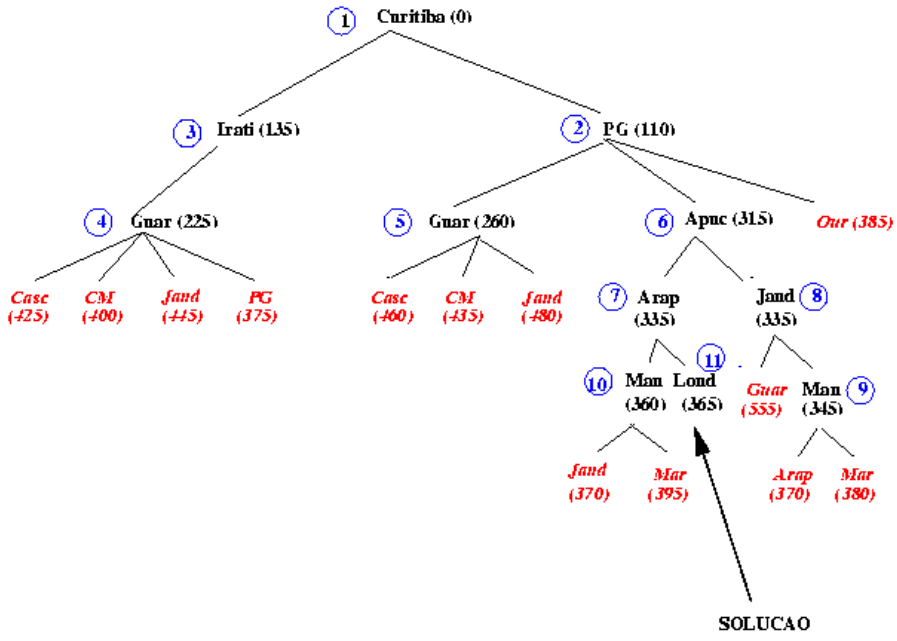
9

Com relação ao uso de memória, a BFS não costuma ser a primeira opção para dispositivos que possuem limitações nesse quesito, uma vez que no pior caso pode precisar expandir todos os nós até encontrar uma solução. Considerando um dispositivo que dispõe de pouca memória e um jogo que possua uma quantidade alta de nós a ser avaliados na busca do caminho até objetivo, podem ocorrer travamentos ou *delay* na sua execução.

3.6.2. Busca de Custo Uniforme

Semelhante à busca em largura, entretanto a Busca de Custo Uniforme expande o nó com o caminho de custo mais baixo (Ilustração 32). Por este motivo, não é recomendada para problemas onde possa haver ações nulas, neste caso o algoritmo poderá ficar paralisado em um laço de repetição infinito. Respeitando essa regra, é **completa** e **ótima,** uma vez que expande os nós em ordem crescente de custo de caminho, retornando assim o caminho de menor custo. (RUSSEL, 2003)

Ilustração 32 – Exemplo de execução do algoritmo de custo uniforme.



Fonte: http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/IA/ResolProb/resproblema.html

No protótipo criado para exemplificar este algoritmo, a função custo foi definida assim:

```
var cost = ob.cost+1;
```

onde **ob.cost** é o custo do caminho do nó inicial até o nó pai do nó atual. O custo do nó atual é o custo do seu nó pai acrescido de 1, que representa a distância do pai até o nó atual.

3.6.3. Best First Search (Busca Gulosa)

No modelo de Busca em Largura, o algoritmo por não fazer ideia de onde o objetivo estava, expandia todos os nós em todos os sentidos. Em teoria, ele encontra o objetivo, no entanto, em um jogo, isso pode não ser suficiente. Por explorar tantos nós, esse algoritmo tende a ser dispendioso, o que pode provocar picos de lentidão.

Por isso, nessa outra versão do protótipo será usada a **Best First Search**, que faz parte da classe de algoritmos de busca informada, a qual introduz o conceito de custo.

Para o cálculo do custo, baseia sua execução na seguinte equação

$$f(n) = g(n)$$

onde f(n) representa o custo estimado para que o objetivo seja alcançado. g(n) é calculado baseado em uma função **heurística**, que estima o custo do caminho do nó *n* até o nó objetivo.

A escolha dessa função heurística deve ser baseada no conhecimento do problema específico ao qual será aplicada. Deve ser também **admissível**, isto é, não deve superestimar o custo real da solução.

No presente trabalho, para a execução do algoritmo **Best Firs Search**, foi utilizada a heurística abaixo:

```
cost = Math.abs(x-targetx)+Math.abs(y-targety);
```

Como pode-se perceber, trata-se de uma heurística admissível, uma vez que a menor distância entre dois pontos, é uma linha reta.

Através da heurística foi desenvolvido um algoritmo de **Busca Gulosa**, que expande o nó aparentemente mais próximo ao nó final com base no valor obtido pela heurística. É **completo** (se souber detectar estados repetidos, caso constrário pode entrar em *loop* infinito), mas **não é ótimo**.

3.6.4. A*

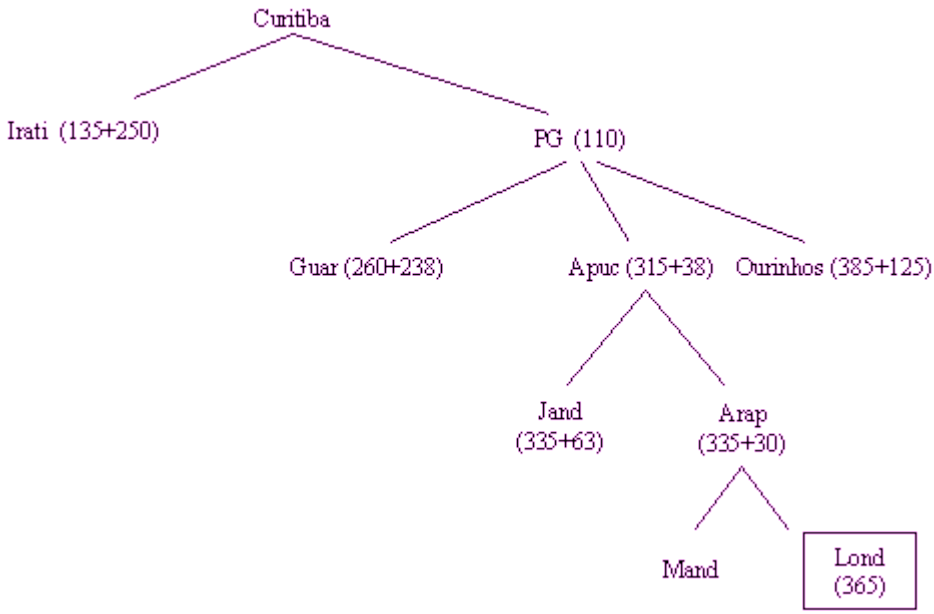
O A* é um dos algoritmos de busca de caminho mais utilizado atualmente. Combina as técnicas da Busca Gulosa (econômica, mas não é ótima e dependendo de sua implementação, não é completa) e da Busca de Custo Uniforme (algumas vezes ineficiente, porém completa e ótima) (LEE, 2013).

É uma estratégia de busca informada (Ilustração 33) , que classifica os nós a serem expandidos seguindo a heurística:

$$f(n) = g(n) + h(n)$$

Onde **f(n)** (custo total estimado do caminho que inicia em *S*, passa por *n* e atinge o nó final) é definido por **g(n)** (distância de *n* até o nó inicial *S*) acrescido de **h(n)** (valor retornado pela heurística aplicada que corresponde ao valor estimado do caminho de *n* até o nó objetivo).

Ilustração 33 - Exemplo de execução do algoritmo A*



Fonte: http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/IA/ResolProb/BuscaBestFirstLondrina.gif

O algoritmo trabalha classificando os nós segundo o valor de f(n), expandindo o nó com o menor custo.

No protótipo criado para testar este método, foi escolhida a **Distância Manhattan** como heurística, que é admissível e fácil de ser obtida. Desta forma o cálculo de f(n) deu-se por:

```
var costG = ob.costG+1;

var costH = Math.abs(x-targetx)+Math.abs(y-targety);

var costF = costG+costH;
```

Cada nó armazena seu valor g(n), que pode mudar caso haja um caminho mais curto até *n* que o avaliado previamente e f(n), utilizado para classificar os nós e expandi-los segundo o menor custo.

4 EXPERIMENTOS E RESULTADOS

4.1. Experimentos

Conforme descrito no capítulo anterior, foram desenvolvidos duas versões do jogo **Territory** e, para cada uma dessas versões foram testados os seguintes algoritmos de busca de caminho: Busca em Largura, Busca de Custo Uniforme, Best First Search (Busca Gulosa) e A*.

Em cada execução foram observadas as seguintes características: número de nós expandidos, número de nós no caminho solução e uso de memória. Para a análise das duas primeiras características foi utilizada a saída dos próprios protótipos e para a medição do uso de memória foi usado o software *Adobe Device Central,* que é distribuído juntamente com o *Adobe Flash*, o qual foi usado para o desenvolvimento dos protótipos usando a linguagem de script *Action Script 2*.

4.2. Medidas de Comparação

Completeza: se existir uma solução, o algoritmo é capaz de encontrá-la?

Otimização: a estratégia encontra a solução ótima, ou seja, aquela que tem o menor custo de caminho dentre todas as soluções?

Complexidade de Tempo: quanto tempo é necessário até que encontre uma solução? Geralmente é medido em termos do número de nós gerados durante a busca.

Complexidade de Espaço: quanta memória é necessária para a execução satisfatória do algoritmo? É medido baseando-se no número máximo de nós armazenados na memória.

A complexidade de tempo e espaço são sempre consideradas em relação a alguma medidade de dificuldade específica do problema (RUSSEL, 2003).

A complexidade é expressa da seguinte forma: número máximo de sucessores de qualquer nó (**b**), profundidade do nó objetivo menos profundo (**d**) e comprimento máximo de qualquer caminho no espaço de estados (**m**).

4.3. Resultados

Para medir o desempenho dos dois protótipos construídos, foram analisados três pontos: número de nós expandidos, número de nós no caminho solução e uso de memória. A seguir serão apresentados os resultados obtidos.

Número de nós expandidos

O número de nós expandidos por cada um dos algoritmos testados foi analisado levando em consideração:

- Protótipo 1: nó inicial, variável entre (0,0) e (0,11) e nó final variável entre (2,14), (6,14) e (10,14)
- Protótipo 2: nó inicial, variável entre (0,0) e (0,23) e nó final variável entre (3,28), (12,28) e (22,28)

Os dados obtidos (Apêndice A do presente trabalho) foram convertidos para gráficos a fim de obter uma melhor visualização dos mesmos.

Ilustração 34 – Gráfico de comparação do número de nós expandidos para o nó final (2,14) : Protótipo 1

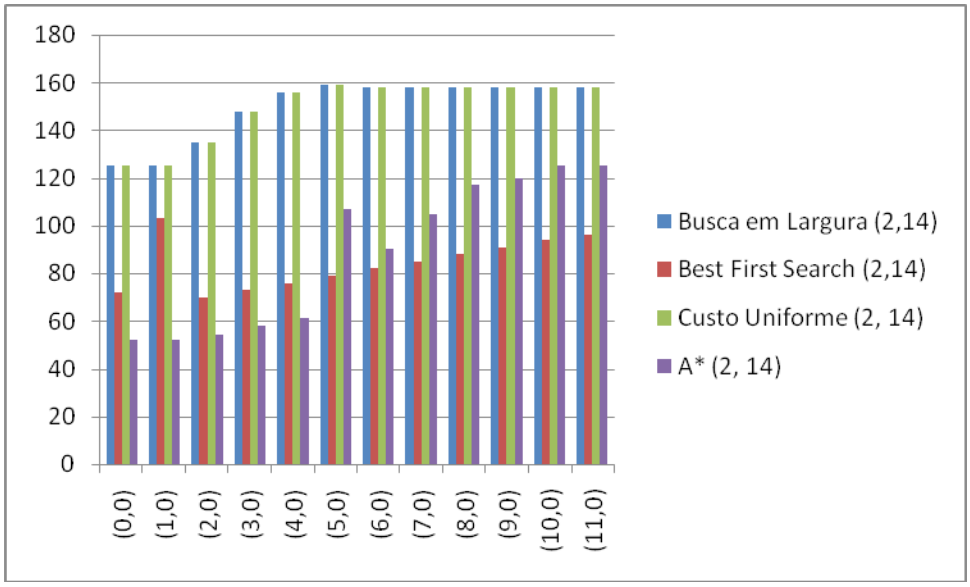


Ilustração 35 – Gráfico de comparação do número de nós expandidos para o nó final (6,14) : Protótipo 1

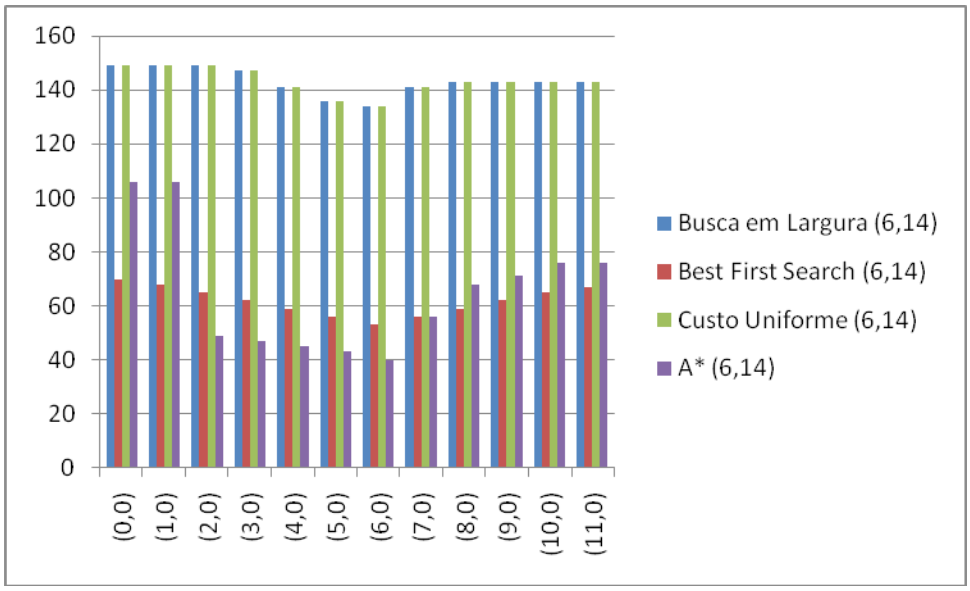


Ilustração 36 – Gráfico de comparação do número de nós expandidos para o nó final (10,14): Protótipo 1

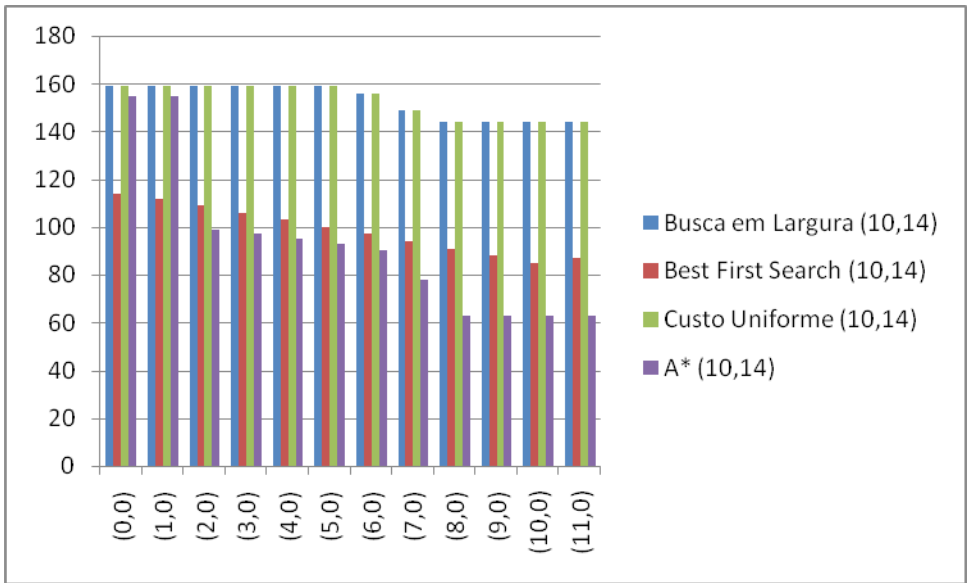


Ilustração 37 – Gráfico de comparação do número de nós expandidos para o nó final (3,28) : Protótipo 2

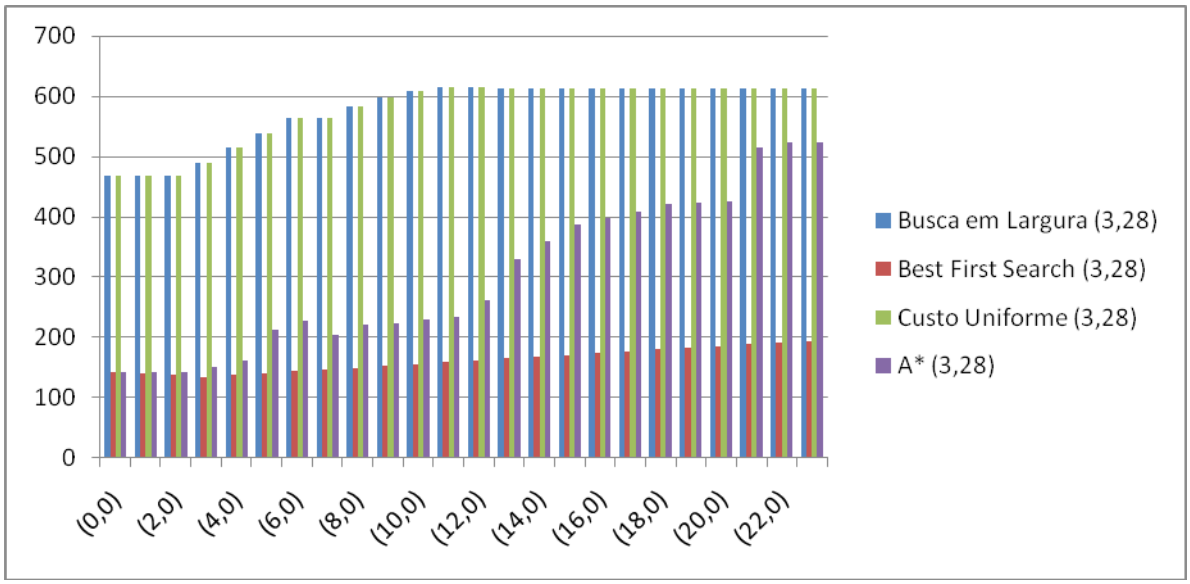


Ilustração 38 – Gráfico de comparação do número de nós expandidos para o nó final (12,28) : Protótipo 2

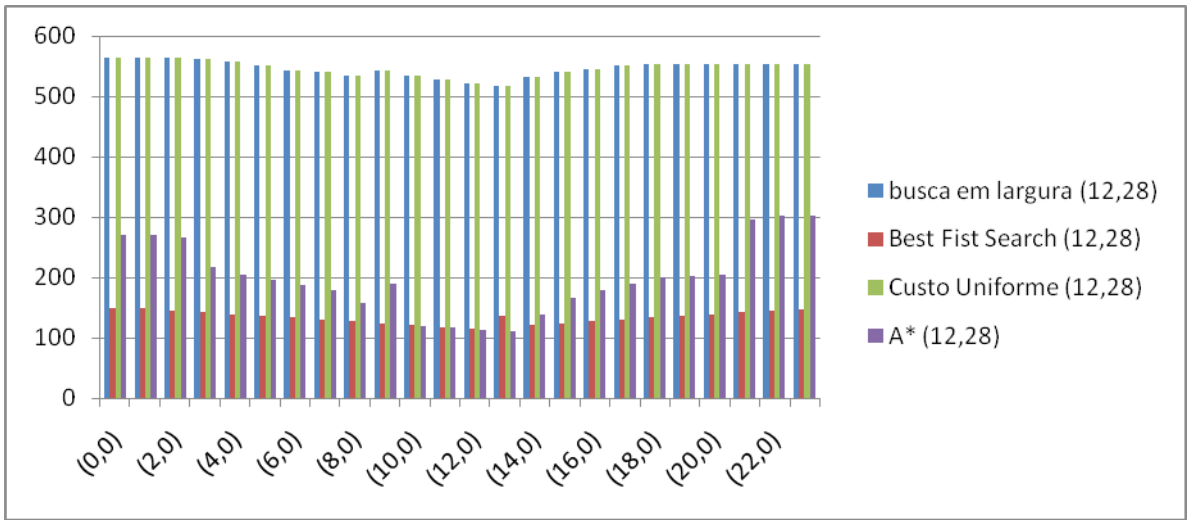
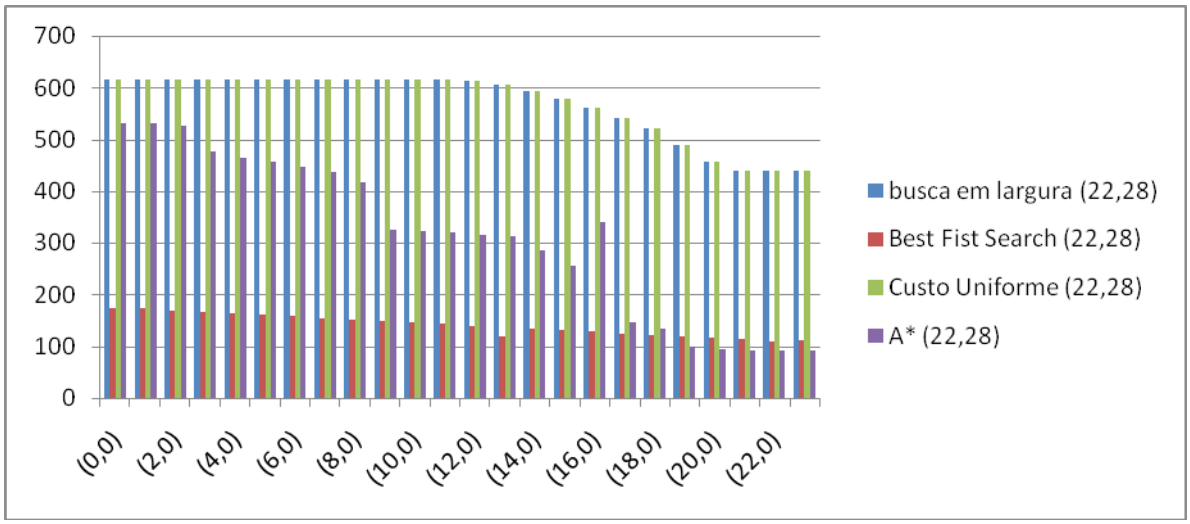


Ilustração 39 – Gráfico de comparação do número de nós expandidos para o nó final (22,28) : Protótipo 2



Como pode ser observado nas Ilustrações 34 a 39, a Busca em Largura e a Busca de Custo Uniforme são os algoritmos que expandem mais nós para que possam encontrar o caminho solução. Pode-se perceber também que, na maioria das vezes, a Best First Search (Busca Gulosa) expande um número menor de nós que o algoritmo A*. Entretanto, há algumas excessões, como nos casos dos nós iniciais (19,0), (20,0), (21,0), (22,0) e (23,0) na Ilustração 39.

Número de nós no caminho solução

O número de nós no caminho solução foi outra grandeza observada durante a execução dos algoritmos, para que sua otimalidade fosse medida. As tabelas obtidas durante a execução dos protótipos e usadas para a elaboração dos gráficos podem ser consultadas no Apêndice B deste trabalho.

Ilustração 40 – Gráfico de comparação do número de nós no caminho solução para o nó final (2,14) : Protótipo 1

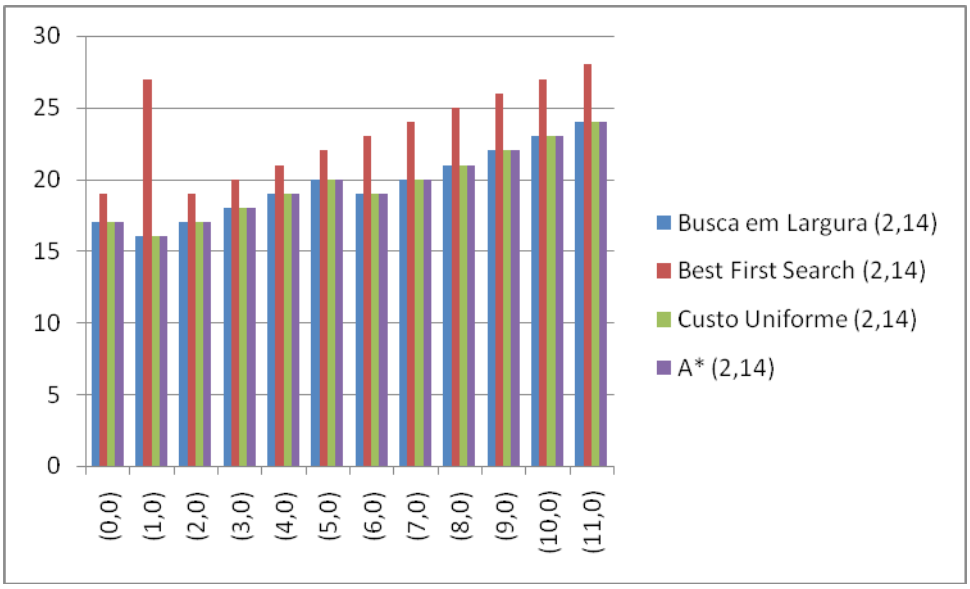


Ilustração 41 – Gráfico de comparação do número de nós no caminho solução para o nó final (6,14) : Protótipo 1

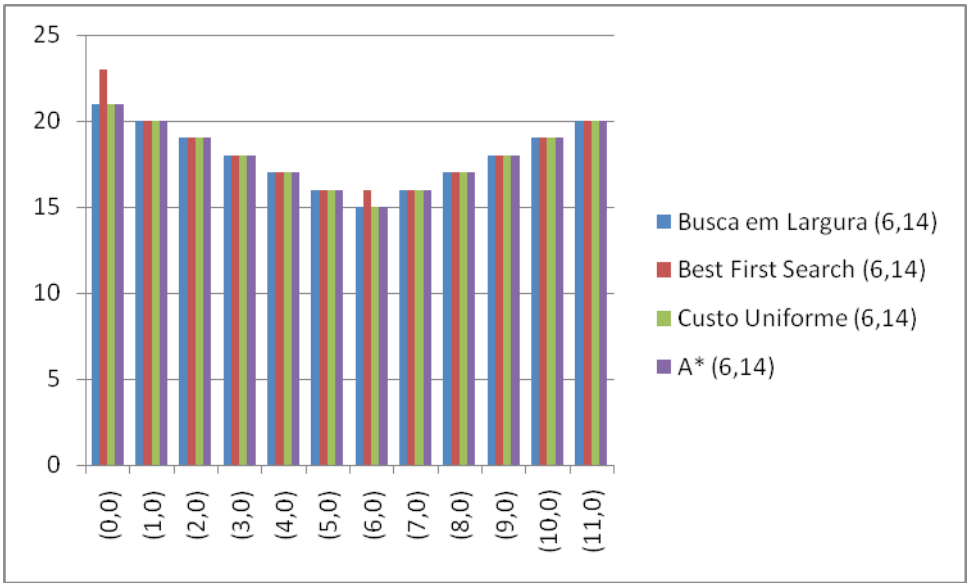


Ilustração 42 – Gráfico de comparação do número de nós no caminho solução para o nó final (10,14) : Protótipo 1

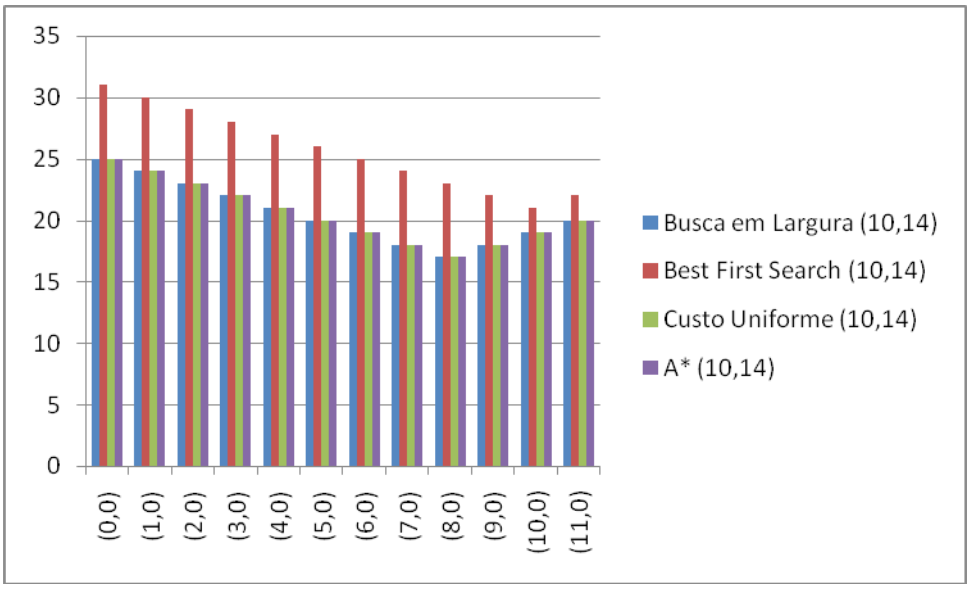


Ilustração 43 – Gráfico de comparação do número de nós no caminho solução para o nó final (3,28) : Protótipo 2

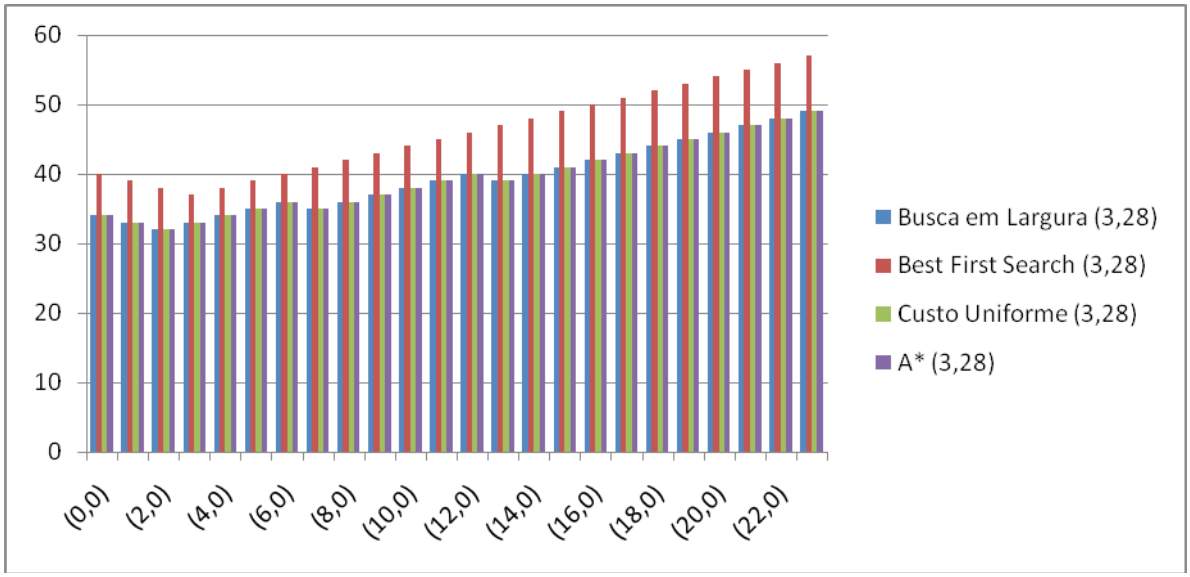


Ilustração 44 – Gráfico de comparação do número de nós no caminho solução para o nó final (12,28) : Protótipo 2

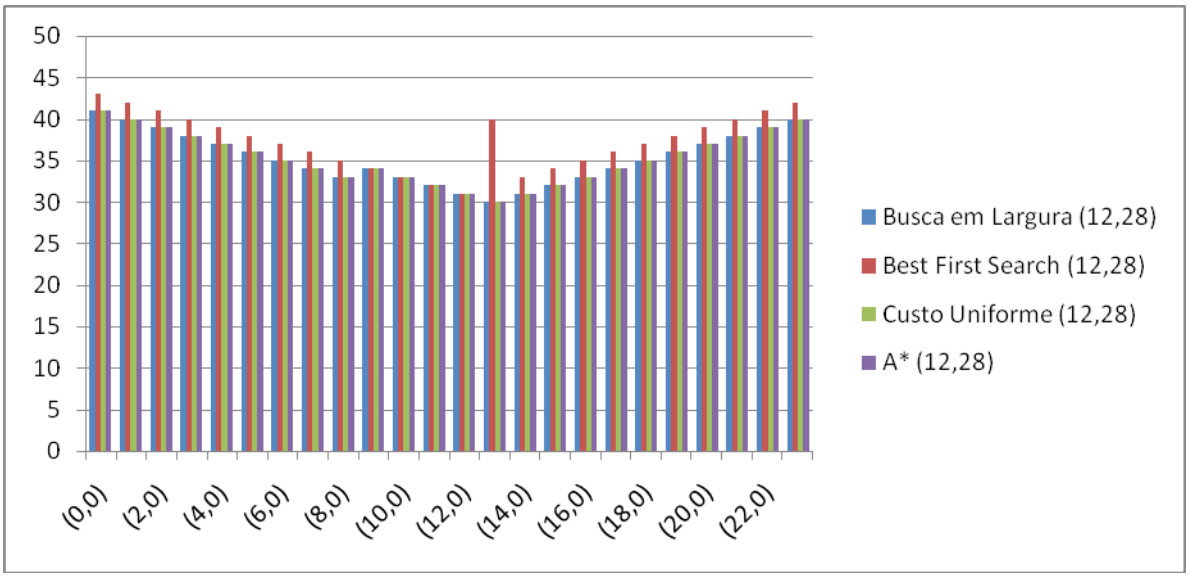
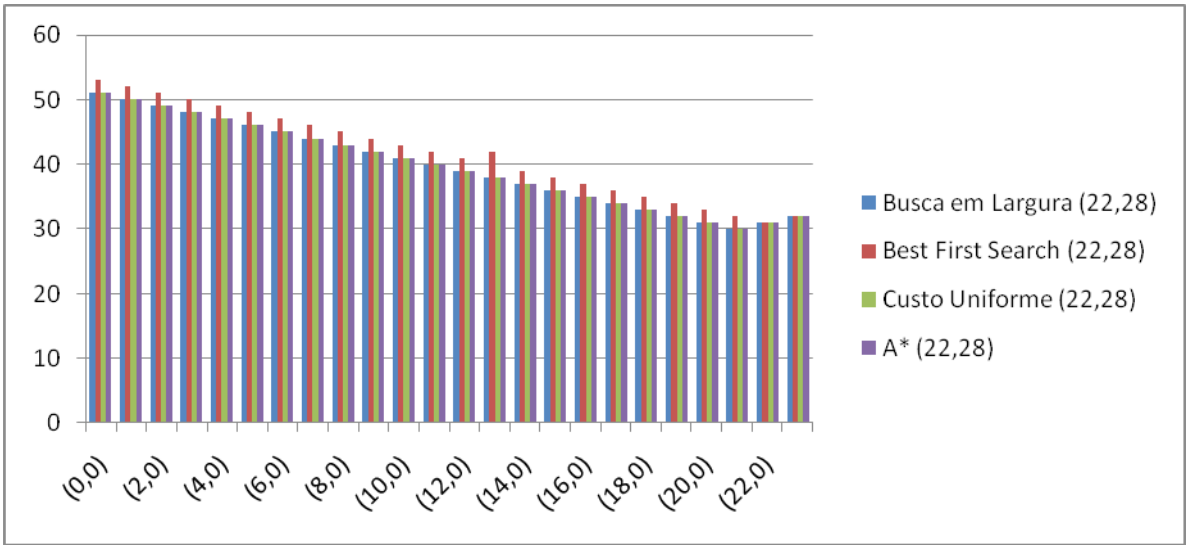


Ilustração 45 – Gráfico de comparação do número de nós no caminho solução para o nó final (22,28) : Protótipo 2



Como pode ser obsevado nas Ilustrações 40 a 45, a Busca em Largura, Busca de Custo Uniforme e A* obtém o resultado ótimo para a busca, ou seja, encontram o menor caminho entre o nó inicial e o nó objetivo. Pode-se perceber também que, a Best First Search (Busca Gulosa) não é ótima, ou seja, não há grantias de que sempre encontrará o menor caminho solução. Entretanto, para alguns casos a Busca Gulosa encontra o caminho ótimo, assim como os demais algoritmos testados. Como exemplo, pode-se observar os casos dos nós iniciais

(9,0), (10,0), (11,0), e (12,0) na Ilustração 44 e (22,0) e (23,0) na Ilustração 45 e na maioria dos nós iniciais da Ilustração 41.

Uso de memória

Para fins de medição de uso de memória, foi utilizado o software *Adobe Device Central,* que visa proporcionar aos desenvolvedores de conteúdos móveis maneiras de testar suas aplicações em uma grande variedade de dispositivos que podem ser emulados através de sua interface.

Através deste software é possível interagir com os dispositivos emulados como da mesma maneira como com dispositivos reais e testar os níveis de desempenho, os estados da rede, a memória, os níveis de potência da bateria e, até mesmo, tipos de iluminação.

O Device Central oferece uma biblioteca de dispositivos, onde para cada dispositivo, o desenvolvedor pode obter informações como os tipos de mídia e de conteúdo suportados.

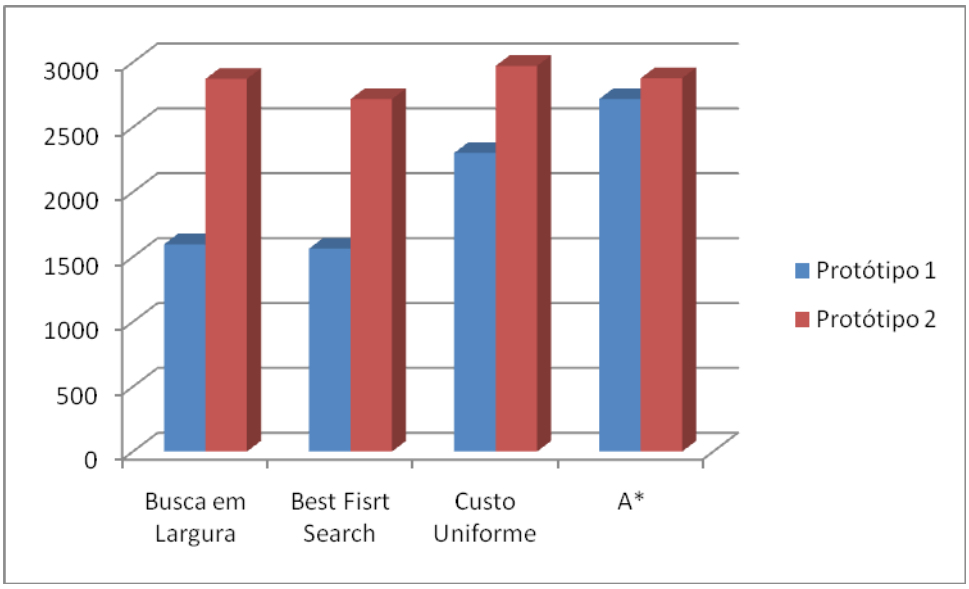
Para testar os protótipos desenvolvidos neste trabalho foi escolhido o seguinte dispositivo emulado pelo Adobe Device Central:

- Flash Lite 3.0, 32 px, 240x320 - O nome do dispositivo representa: a versão do *flash lite*, tipo de pixelagem e tamanho da tela. Este dispositivo possui as seguintes configurações de memória: 512kB de *Heap* estático, 16384kB de *Heap* Dinâmico e 256kB de Armazenamento Persistente.

Para cada execução foram retirados dois *snapshots* para demonstrar o uso de memória das aplicações no emulador. O primeiro *snapshot* corresponde a inicialização da aplicação no emulador e o segundo ao pico de uso de memória durante a execução. As imagens dos snapshots podem ser visualizadas no Apêndice C deste trabalho.

Os dados da memória utilizada no pico de uso de memória durante sucessivas execuções dos protótipos foram organizados em um gráfico, como pode ser visto na Ilustração 46.

Ilustração 46 – Gráfico de comparação do pico de uso de memória dos dois protótipos desenvolvidos



Conforme a Ilustração 46, pode-se perceber que o algoritmo Best First Search é o mais econômico em termos de uso de memória, quando executado em ambos os protótipos. Entretanto, o algoritmo que utiliza uma maior quantidade de memória não se mantém nos dois protótipos: na execução do Protótipo 1, o algoritmo A* foi o que usou mais memória e no Protótipo 2, o algoritmo que usou mais memória foi o Busca de Custo Uniforme.

4.3.Discussão

Quanto à **completeza** e **otimização**, os algoritmos desenvolvidos obtiveram o comportamento esperado, ou seja, todos são completos e apenas a Best First Search (Busca Gulosa) não é ótima.

Dos algoritmos testados, os que expandem mais nós em sua busca foram Busca em Largura e Busca de Custo Uniforme, o que não os indica para uso em aplicações onde o espaço de busca seja muito grande, pois podem causar lentidão na aplicação.

Apesar de, o algoritmo Best First Search (Busca Gulosa) não ser ótimo, ou seja, não garante que o caminho de menor custo seja encontrado, a diferença apresentada nos caminhos encontrados, não pode ser considerada exorbitante e, em algumas execuções o algoritmo consegue devolver o caminho ótimo.

Quanto ao **uso de memória**, como podemos ver na Ilustração 46, os algoritmos que trabalham com busca informada utilizam mais memória, isto deve-se ao fato de que cada *tile* expandido precisa armazenar o valor de seu custo, o que não era necessário em algoritmos baseados em busca não informada. Também é possível verificar que, algumas vezes a execução dos algoritmos de busca de caminho chega a ocupar 17% da memória do aparelho (no cado do aparelho simulado para os testes, que apresentava 512kB de *Heap* estático, 16384kB de *Heap* Dinâmico e 256kB de Armazenamento Persistente), o que é um valor elevado, visto que, na execução da aplicação em um dispositivo real, haverá mais aplicações disputando o uso da memória, o que poderá causar travamentos e/ou lentidão na execução do jogo.

O uso do método *tile* para compor os gráficos do jogo colaboraram para que o tamanho da aplicação ficasse entre 128KB e 135KB e, ainda assim fosse possível criar uma identificação visual adequada, onde é possível facilmente identificar as áreas que compõe o jogo, como água, grama e terra.

5 CONSIDERAÇÕES FINAIS

O desenvolvimento de jogos está atingindo diversas plataformas. Muitas dessas utilizam recursos limitados. É o caso de celulares, *tablets* e jogos executados em *browsers.*

A maioria dos jogos que atualmente foram desenvolvidos para ser executados em plataformas com baixos recursos de *hardware*, não conta com algoritmos elaborados de busca de caminho.

A programação dos NPCs utilizando *scripts* fixos e movimentos aleatórios são bastante difundidos, pois usam poucos recursos da máquina, mas podem tornar algumas rotinas do jogo repetitivas e, em alguns casos isso pode prejudicar a jogabilidade do mesmo. Este é o caso do jogo *Super Bomberman*[10] produzido pela *Living Mobile*, onde o jogo usa o método *Tile* para a descrição do cenário. Entretanto, a inteligência dos inimigos do agente principal é baseada em *scripts* e movimentos aleatórios.

No caso de jogos e aplicações complexas, não se pode concluir se o fato dos *scripts* fixos serem preferidos em vez dos algoritmos inteligentes se deve ao fato de não se poder garantir uma execução aceitável em plataformas com recursos limitados, uma vez que, conforme os resultados obtidos neste trabalho, provou-se aplicável o uso de algoritmos inteligentes em jogos simples.

Tonypa (2003 – 2005) desenvolveu dois algoritmos de busca de caminho (B*readth-First Search e Best-First Search*), mas não apresenta comparações contundentes entre eles. Também não apresenta algoritmos baseados em busca informada, como o A* que geralmente obtém melhores resultados nas condições de busca de caminho.

No presente trabalho, foi desenvolvido um jogo - **Territory** - para testar alguns algoritmos de busca de caminho e verificar sua aplicabilidade em ambientes que rodam em hardwares com recursos limitados. Para desenvolvimento dos gráficos do mesmo, foi utilizado o método *Tile*, para que fosse possível obter uma melhor qualidade gráfica e resultando em uma aplicação que usa uma quantidade menor de memória para seu armazenamento. Territory foi desenvolvido em duas versões, conforme o tamanho dos *tiles* que o compõe (Protótipo 1 - 12x15 *tiles* e Protótipo 2 - 24x29 *tiles*). A escolha de desenvolvimento em duas versões deve-se à aplicação dos testes de algoritmos de busca de caminho sobre *grids* de tamanhos diferentes, observando a quantidade de memória utilizada para a execução dos algoritmos, assim como, possíveis travamentos causados quando essa quantidade utilizada é muito alta.

---

[10] Disponível em: *http:// livingmobile.net*

Nas execuções dos protótipos desenvolvidos, foi possível perceber a possibilidade de reproduzir os algoritmos de busca de caminho em plataformas com recursos limitados sem que a jogabilidade seja prejudicada. Foi utilizado o método *tile*, visando obter, ao mesmo tempo, uma qualidade gráfica aceitável e uma economia no uso de recursos do aparelho que irá executar a aplicação. O mesmo *grid* utilizado para o mapeamento da imagem de fundo do jogo para aplicação do método *Tile* foi utilizado para desenvolver os algoritmos de busca de caminho aplicados aos inimigos do jogador.

Entretanto, o próprio método *Tile* impossibilitou o desenvolvimento de um *grid* maior para testes, pois ao diminuir o tamanho dos *tiles,* a jogabilidade era prejudicada, uma vez que a movimentação dos personagens é feita sobre os *tiles* e, se eles tiverem um tamanho muito reduzido o gráfico do jogo fica desconfigurado e induz movimentos artificiais. Como exemplo, com *tiles* menores que 10x10 px, quando os inimigos circulam uma região com água, devido ao seu tamanho maior que o *tile*, ao jogador pode parecer como se o *tile* fosse andável, o que não é a realidade.

Apesar de todos os métodos testados terem uma execução aceitável do ponto de vista do jogador (não há travamentos durante o jogo), foi possível perceber que, a escolha de qual algoritmo de busca de caminho utilizar é de muita importância no desenvolvimento de um jogo. Isso se deve ao fator diversão que o mesmo deve proporcionar ao usuário da aplicação.

Três dos algoritmos testados (Busca em Largura, Busca de Custo Uniforme e A*) tem desempenho **ótimo**, ou seja, encontram o menor caminho entre o ponto inicial e o ponto objetivo. Entretanto, esse comportamento ótimo acaba por tornar previsível o caminho a ser percorrido pelos inimigos durante o jogo, o que pode parecer vantajoso, mas pode acabar por diminuir o tempo em que o jogador se sente "entretido" jogando.

Assim, como trabalhos futuros propõe-se: a melhoria dos algoritmos de detecção de colisão dos inimigos, para que o tamanho dos *tiles* utilizados na criação do campo de jogo possa ser diminuído, o que pode gerar um comportamento diferente quanto a execução sem travamentos por parte dos algoritmos testados; implementar um nível maior de inteligência nos inimigos focada na tentativa de "ganhar" do jogador e, não apenas atingir seu objetivo; modificação dos algoritmos de busca de caminho de forma a gerar um caminho menos artificial e previsível.

6 REFERÊNCIAS

AIOLLI, Fabio; PALAZZI, Claudio E..**Enhancing Ar t ificial Intelligence on a Re al Mobile Game.** Department of Pure and Applied Mathematics, University of Padova, Itália, 2008.

AUGUSTO, F. V. R. Dubina, A. B. **Inteligência Artificial nos Jogos.** 2009. 22 f. Disponível em: <http://www.slideshare.net/fernandoovargas/artigo-inteligencia-artificaljogos>. Acesso em 5 mai. 2012.

BONFANDINI, E. **Algoritmo de Busca e Planejamento de Caminhos *Pathfinding*. (2011)** Disponível em: <http://nostaljia.eng.br/beatnupproject/wp-content/uploads/artigo_path_finding_eduardo_bonfandini.pdf>. Acesso em 14 abr. 2012.

CALLOIS, R. **Los juegos y los hombres: la máscara y el vértigo.** 1 ed. México: Fondo de cultura económica, 1986.

DUH, Henry Been-Lirn; CHEN, Vivian Hsueh Hua; TAN, Chee Boon. **Playing Different Games on Different Phones: An Empirical Study on Mobile Gaming.** Proceedings of the 10th international conference on Human computer interaction with mobile devices and services, New York, NY, USA,2008. Disponível em <http://dl.acm.org/citation.cfm?id=1409296> Acesso em julho de 2013.

EXECUTIVE. **Tipos de Ambientes em Inteligência Artificial.** Disponível em: <http://www.executive.com.br/inteligencia-artificial/>. Acesso em 14 abr. 2012.

FEOFILOFF, Paulo. **Algoritmos para grafos em C.** Disponível em <http://www.ime.usp.br/~pf/algoritmos_para_grafos/index.html>. Acesso em setembro de 2013.

FERREIRA, P. R. L. **Movimento e Pathfinding usando Navigation Meshes em um Cenário Interativo.** 2009. 35 f. Monografia de projeto orientado em computação II - Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, 2009.

FLAUSINO, Rodrigo. **Estado da Arte da Inteligência Artificial para jogos Eletrônicos.** 2007. Disponível em: <http://www.rodrigoflausino.com.br/artigos-e-tutoriais/estado-da-arte-da-inteligencia-artificial-para-jogos-eletronicos/> Acesso em novembro de 2012.

FLEXA, P. **Inteligência Artificial: Pathfinding.** Disponível em: <http://www.sharpgames.net/Artigos/Artigo/tabid/58/selectmoduleid/376/ArticleID/1663/reftab/36/Default.aspx>. Acesso e 15 mai. 2012.

GAGNON, Michel. **Resolução de Problemas.** Disponível em <http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/IA/ResolProb/resproblema.html>. Acesso em agosto de 2013.

GAYOSO, Reinaldo N. **A* Pathfinding para iniciantes.** Por Patrick Lester (Atualizado em 21 de abril de 2004)
(Traduzido por Reynaldo N. Gayoso, rngayoso@msn.com em 04 de Abril de 2005)
(Atualizado em 04 de junho de 2005)

GRUNDVIG, M. **Fast PathFinding in Flash.** Disponível em: <http://www.electrotank.com/junk/mike/ai/PrecomputedPathTutorial.html>. Acesso em 15 mai. 2012.

JUNIOR, Gilberto Timótheo. LIMA, Sérgio M. B. **Algoritmos genéticos aplicados a jogos eletrônicos.** Disponível em <> Acesso em janeiro de 2013.

KARLSSON, Borje Felipe Fernandes. **Um middleware de Inteligência Artificial para games.** Rio de Janeiro, 2005. Disponível em <> Acesso em dezembro de 2012.

KISHIMOTO, A. **Inteligência Artificial em Jogos Eletrônicos.** 2004. 11 f. Disponível em: <http://www.programadoresdejogos.com/trab_academicos/andre_kishimoto.pdf>. Acesso em: 10 mai. 2012.

KOLLER, Alex. **Java ME vs. Flash Lite: A comparison of mobile phone game development.** 2007. 75 f. Submitted in partial fulfilment of the requirements of the degree of Bachelor of Science (Honours) of Rhodes University, Grahamstown, South Africa, 2007.

LEE, Huei Diana. **Inteligência Artificial: Estratégias de Busca Informada.** Disponível em <http://dainf.ct.utfpr.edu.br/~fabro/IA_I/busca/IA_Estrategias_Busca_Inf.pdf> Acesso em setembro de 2013.

LESTER, P. **A * Pathfinding para Iniciantes.** Disponível em: <http://www.policyalmanac.org/games/aStarTutorial_port.htm>. Acesso em 16 mai. 2012.

LIU, Qiang; DIAO, Lixin; Tu, Guangcan. **The application of Artificial Intelligence in Mobile Learning**. International Conference on System Science, Engineering Design and Manufacturing Informatization, 2010. Disponível em <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05640149> Acesso em junho de 2013.

LOPES, G. **Jogos Eletrônicos: conceitos gerais.** Disponível em: <http://www-usr.inf.ufsm.br/~pozzer/disciplinas/cga_8_classificacao_jogos.pdf>. Acesso em 5 mai. 2012.

LOUREIRO, C. B.; SILVA, A. F. da. **Inteligência Artificial e sua aplicabilidade nos jogos.** Disponível em: <http://guaiba.ulbra.tche.br/pesquisa/2009/artigos/sistemas/salao/482.pdf>. Acesso em 18 mai. 2012.

MACHADO, A. F. V. **Inteligência Computacional para Jogos Eletrônicos.** Disponível em: <www.riopomba.ifsudestemg.edu.br/dcc/dcc/materiais/1063182433_slide-1-introducao.ppt>. Acesso em 10 mai. 2012.

NEGRINE, A. O lúdico no contexto da vida humana: da primeira infância à terceira idade. In: SANTOS, S. M. P dos. (org.). **Brinquedoteca: a criança, o adulto e o lúdico.** 6 ed. Petrópolis: Vozes, 2000.

PASTOR, Thiago Dias; CORRÊA, Bruno Duarte. **Navegação em jogos eletrônicos usando PathFinding e Steering Behavior**. Escola Politécnica da Universidade de São Paulo, São Paulo, 2010.

POZZER, C. T. Planejamento de Caminho. 2006. Disponível em: <hhtp://*www-usr.inf.ufsm.br/~pozzer/disciplinas/pj3d_path.pdf* >. Acesso em 20 mai. 2012.

RIBEIRO, Rafael Dias. **Inteligência Computacional**. Disponível em <http://www.rafaeldiasribeiro.com.br/downloads/IC1_7.pdf> Acesso em setembro de 2013.

RUSSEL, S.; NORVIG, P. **Inteligencia Artificial uma Abordagem Moderna.** 2 ed. São Paulo: Campus, 2003.

SANTANA, R. T. de. **I.A. em Jogos: a Busca Competitiva Entre o Homem e a Máquina.** 2006. Monografia para Tecnólogo em Informática com ênfase em Gestão de Negócios, Faculdade de Tecnologia de Praia Grande, Praia Grande, 2006.

SCHWAB, B. **AI Game Engine Programming**, 1 ed. Hingham, Massachusetts: Charles River Media, 2004.

SEDGEWICK, Robert. **Algorithms in C (part 5: Graph Algorithms).** 3 ed. Addison-Wesley/Longman, 2002.

SILVA, D. A. de O. **Inteligência Artificial em Jogos.** Disponível em: <http://www.dsc.ufcg.edu.br/~pet/ciclo_seminarios/tecnicos/2007/GameIA.pdf>. Acesso em 3 mai. 2012.

SIMÕES. Alexandre da Silva. **Aula 05 – Busca com informação**. Disponível em <http://www.gasi.sorocaba.unesp.br/assimoes/lectures/iac/downloads/busca_heuristica.pdf>.Acesso em abril de 2013.

SOUZA, MARCELO. **Influência dos jogos no campo da inteligência artificial**. Disponível em <http://ebookbrowsee.net/influencia-dos-jogos-no-campo-da-inteligencia-artificial-marcelo-de-souza-pdf-d489394670> Acesso em janeiro de 2013.

TATAI, V. K. **Técnicas de Sistemas Inteligentes Aplicadas ao Desenvolvimento de Jogos de Computador.** 2006. 129 f. Dissertação de Mestrado em Engenharia Elétrica, Departamento de Engenharia de Computação, Faculdade de Engenharia Elétrica e de COmputação, Universidade Estadual de Campinas, Campinas, 2006.

TONYPA. **Tile Based Games** (2005) Disponível em: <http://www.tonypa.pri.ee/tbw/>. Acesso em 10 abr. 2012.

VIEIRA FILHO, Vicente. **Revolution AI Engine: Desenvolvimento de um motor de Inteligência artificial para criação de jogos eletrônicos.** Acesso em dezembro de 2012.

VIEIRA, V. **Revolution AI Engine: Desenvolvimento de um Motor de Inteligência Artificial para a Criação de Jogos Eletrônicos.** 2005. 75 f. Trabalho de Conclusão de Curso, Graduação em Ciência da Computação - Centro de Informática, Universidade Federal de Pernambuco, Recife, 2005.

XAVIER, LAIS. **Um Estudo Comparativo Entre Flash Lite e J2ME para Desenvolvimento de Jogos**. Disponível em <www.cin.ufpe.br/~tg/2006-2/lx.pdf>. Acesso em 18 de setembro de 2012.

XIN, Chen. **Artificial Intelligence Application in Mobile Phone Serious Game.** First International Workshop on Education Technology and Computer Science, 2009. ETCS 2009. Wuhan University. Wuhan, China, 2009.

XU, Zhiguang; DOREN Michael Van. **A Museum Visitors Guide with the A* Pathfinding Algorithm**. Valdosta State University. Valdosta, GA, USA, 2011. Disponível em <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05953171> Acesso em agosto de 2013.

YAMAMOTO, F.S. **Inteligência Artificial em Jogos Eletrônicos Interativos.** Universidade de São Paulo – EPUSP, 2002.

ZUBEN, F. J. V. **Estruturas e Estratégias de Busca.** Disponível em: <ftp://ftp.dca.fee.unicamp.br/pub/docs/vonzuben/ea072_2s06/notas_de_aula/topicoP2.4_06.pdf>. Acesso em 11 mai. 2012.

APÊNDICE A: Número de nós expandidos

Tabela 2 - Número de nós expandidos: Protótipo 1 - Busca em Largura

Nó incial/ Nó final	(2,14)	(6,14)	(10,14)
(0,0)	125	149	159
(1,0)	125	149	159
(2,0)	135	149	159
(3,0)	148	147	159
(4,0)	156	141	159
(5,0)	159	136	159
(6,0)	158	134	156
(7,0)	158	141	149
(8,0)	158	143	144
(9,0)	158	143	144
(10,0)	158	143	144
(11,0)	158	143	144

Fonte: execução do Protótipo 1, com o algoritmo de Busca em Largura

Tabela 3 - Número de nós expandidos: Protótipo 1 - Best Search

Nó incial/ Nó final	(2,14)	(6,14)	(10,14)
(0,0)	72	70	114
(1,0)	103	68	112
(2,0)	70	65	109
(3,0)	73	62	106
(4,0)	76	59	103
(5,0)	79	56	100
(6,0)	82	53	97
(7,0)	85	56	94
(8,0)	88	59	91
(9,0)	91	62	88
(10,0)	94	65	85
(11,0)	96	67	87

Fonte: execução do Protótipo 1, com o algoritmo de Best Search

Tabela 4 - Número de nós expandidos: Protótipo 1 - Custo Uniforme

Nó incial/ Nó final	(2,14)	(6,14)	(10,14)
(0,0)	125	149	159
(1,0)	125	149	159
(2,0)	135	149	159
(3,0)	148	147	159
(4,0)	156	141	159
(5,0)	159	136	159
(6,0)	158	134	156
(7,0)	158	141	149
(8,0)	158	143	144
(9,0)	158	143	144
(10,0)	158	143	144
(11,0)	158	143	144

Fonte: execução do Protótipo 1, com o algoritmo de Custo Uniforme

Tabela 5 - Número de nós expandidos: Protótipo 1 - A*

Nó incial/ Nó final	(2,14)	(6,14)	(10,14)
(0,0)	52	106	155
(1,0)	52	106	155
(2,0)	54	49	99
(3,0)	58	47	97
(4,0)	61	45	95
(5,0)	107	43	93
(6,0)	90	40	90
(7,0)	105	56	78
(8,0)	117	68	63
(9,0)	120	71	63
(10,0)	125	76	63
(11,0)	125	76	63

Fonte: execução do Protótipo 1, com o algoritmo de Busca A*

Tabela 6 - Número de nós expandidos: Protótipo 2 - Busca em Largura

Nó incial/ Nó final	(3,28)	(12,28)	(22,28)
(0,0)	468	564	616
(1,0)	468	564	616
(2,0)	468	564	616
(3,0)	490	562	616
(4,0)	514	558	616
(5,0)	539	552	616
(6,0)	563	543	616
(7,0)	563	540	616
(8,0)	582	533	616
(9,0)	597	543	616
(10,0)	609	534	616
(11,0)	615	528	616
(12,0)	615	521	612
(13,0)	613	517	606
(14,0)	613	532	594
(15,0)	613	540	578
(16,0)	613	545	561
(17,0)	613	552	541
(18,0)	613	553	521
(19,0)	613	553	490
(20,0)	613	553	456
(21,0)	613	553	439
(22,0)	613	553	439
(23,0)	613	553	439

Fonte: execução do Protótipo 2, com o algoritmo de Busca em Largura

Tabela 7 - Número de nós expandidos: Protótipo 2 - Best Search

Nó incial/ Nó final	**(3,28)**	**(12,28)**	**(22,28)**
(0,0)	143	150	175
(1,0)	141	148	173
(2,0)	138	145	170
(3,0)	135	142	167
(4,0)	138	139	164
(5,0)	141	136	161
(6,0)	144	133	158
(7,0)	147	130	155
(8,0)	150	127	152
(9,0)	153	124	149
(10,0)	156	121	146
(11,0)	159	118	143
(12,0)	162	115	140
(13,0)	165	137	118
(14,0)	168	121	134
(15,0)	171	124	131
(16,0)	174	127	128
(17,0)	177	130	125
(18,0)	180	133	122
(19,0)	183	136	119
(20,0)	186	139	116
(21,0)	189	142	113
(22,0)	192	145	110
(23,0)	194	147	112

Fonte: execução do Protótipo 2, com o algoritmo de Best Search

Tabela 8 - Número de nós expandidos: Protótipo 2 - Custo Uniforme

Nó incial/ Nó final	(3,28)	(12,28)	(22,28)
(0,0)	468	564	616
(1,0)	468	564	616
(2,0)	468	564	616
(3,0)	490	562	616
(4,0)	514	558	616
(5,0)	539	552	616
(6,0)	563	543	616
(7,0)	563	540	616
(8,0)	582	533	616
(9,0)	597	543	616
(10,0)	609	534	616
(11,0)	615	528	616
(12,0)	615	521	612
(13,0)	613	517	606
(14,0)	613	532	594
(15,0)	613	540	578
(16,0)	613	545	561
(17,0)	613	552	541
(18,0)	613	553	521
(19,0)	613	553	490
(20,0)	613	553	456
(21,0)	613	553	439
(22,0)	613	553	439
(23,0)	613	553	439

Fonte: execução do Protótipo 2, com o algoritmo de Custo Uniforme

Tabela 9 - Número de nós expandidos: Protótipo 2 - A*

Nó incial/ Nó final	(3.28)	(12,28)	(22,28)
(0,0)	142	270	530
(1,0)	142	270	530
(2,0)	142	267	527
(3,0)	151	216	476
(4,0)	162	205	465
(5,0)	212	196	456
(6,0)	227	187	447
(7,0)	205	178	438
(8,0)	221	157	417
(9,0)	224	189	325
(10,0)	229	119	322
(11,0)	233	116	319
(12,0)	261	113	316
(13,0)	330	110	312
(14,0)	359	139	285
(15,0)	387	167	256
(16,0)	398	178	341
(17,0)	409	189	146
(18,0)	421	201	135
(19,0)	423	203	99
(20,0)	425	205	95
(21,0)	515	295	93
(22,0)	523	303	93
(23,0)	523	303	93

Fonte: execução do Protótipo 2, com o algoritmo de Busca A*

APÊNDICE B: Número de nós no caminho solução

Tabela 10 - Número de nós no caminho solução: Protótipo 1 - Busca em Largura

Nó incial/ Nó final	(2,14)	(6,14)	(10,14)
(0,0)	17	21	25
(1,0)	16	20	24
(2,0)	17	19	23
(3,0)	18	18	22
(4,0)	19	17	21
(5,0)	20	16	20
(6,0)	19	15	19
(7,0)	20	16	18
(8,0)	21	17	17
(9,0)	22	18	18
(10,0)	23	19	19
(11,0)	24	20	20

Fonte: execução do Protótipo 1, com o algoritmo de Busca em Largura

Tabela 11 - Número de nós no caminho solução: Protótipo 1 - Best Search

Nó incial/ Nó final	(2,14)	(6,14)	(10,14)
(0,0)	19	21	31
(1,0)	27	20	30
(2,0)	19	19	29
(3,0)	20	18	28
(4,0)	21	17	27
(5,0)	22	16	26
(6,0)	23	15	25
(7,0)	24	16	24
(8,0)	25	17	23
(9,0)	26	18	22
(10,0)	27	19	21
(11,0)	28	20	22

Fonte: execução do Protótipo 1, com o algoritmo de Best Search

Tabela 12 - Número de nós no caminho solução: Protótipo 1 - Custo Uniforme

Nó incial/ Nó final	(2,14)	(6,14)	(10,14)
(0,0)	17	21	35
(1,0)	16	20	24
(2,0)	17	19	23
(3,0)	18	18	22
(4,0)	19	17	21
(5,0)	20	16	20
(6,0)	19	15	19
(7,0)	20	16	18
(8,0)	21	17	17
(9,0)	22	18	18
(10,0)	23	19	19
(11,0)	24	20	20

Fonte: execução do Protótipo 1, com o algoritmo de Custo Uniforme

Tabela 13 - Número de nós no caminho solução: Protótipo 1 - A*

Nó incial/ Nó final	(2,14)	(6,14)	(10,14)
(0,0)	17	21	25
(1,0)	16	20	24
(2,0)	17	19	23
(3,0)	18	18	22
(4,0)	19	17	21
(5,0)	20	16	20
(6,0)	19	15	19
(7,0)	20	16	18
(8,0)	21	17	17
(9,0)	22	18	18
(10,0)	23	19	19
(11,0)	24	20	20

Fonte: execução do Protótipo 1, com o algoritmo A*

Tabela 14 - Número de nós expandidos: Protótipo 2 - Busca em Largura

Nó incial/ Nó final	**(3,28)**	**(12,28)**	**(22,28)**
(0,0)	34	41	51
(1,0)	33	40	50
(2,0)	32	39	49
(3,0)	33	38	48
(4,0)	34	37	47
(5,0)	35	36	46
(6,0)	36	35	45
(7,0)	35	34	44
(8,0)	36	33	43
(9,0)	37	34	42
(10,0)	38	33	41
(11,0)	39	32	40
(12,0)	40	31	39
(13,0)	39	30	38
(14,0)	40	31	37
(15,0)	41	32	36
(16,0)	42	33	35
(17,0)	43	34	34
(18,0)	44	35	33
(19,0)	45	36	32
(20,0)	46	37	31
(21,0)	47	38	30
(22,0)	48	39	31
(23,0)	49	40	32

Fonte: execução do Protótipo 2, com o algoritmo de Busca em Largura

Tabela 15 - Número de nós expandidos: Protótipo 2 - Best Search

Nó incial/ Nó final	(3,28)	(12,28)	(22,28)
(0,0)	40	43	53
(1,0)	39	42	52
(2,0)	38	41	51
(3,0)	37	40	50
(4,0)	38	39	49
(5,0)	39	38	48
(6,0)	40	37	47
(7,0)	41	36	46
(8,0)	42	35	45
(9,0)	43	34	44
(10,0)	44	33	43
(11,0)	45	32	42
(12,0)	46	31	41
(13,0)	47	40	32
(14,0)	48	33	39
(15,0)	49	34	38
(16,0)	50	35	37
(17,0)	51	36	36
(18,0)	52	37	35
(19,0)	53	38	34
(20,0)	54	39	33
(21,0)	55	40	32
(22,0)	56	41	31
(23,0)	57	42	32

Fonte: execução do Protótipo 2, com o algoritmo de Best Search

Tabela 16 - Número de nós expandidos: Protótipo 2 - Custo Uniforme

Nó incial/ Nó final	**(3,28)**	**(12,28)**	**(22,28)**
(0,0)	34	41	51
(1,0)	33	40	50
(2,0)	32	39	49
(3,0)	33	38	48
(4,0)	34	37	47
(5,0)	35	36	46
(6,0)	36	35	45
(7,0)	35	34	44
(8,0)	36	33	43
(9,0)	37	34	42
(10,0)	38	33	41
(11,0)	39	32	40
(12,0)	40	31	39
(13,0)	39	30	38
(14,0)	40	31	37
(15,0)	41	32	36
(16,0)	42	33	35
(17,0)	43	34	34
(18,0)	44	35	33
(19,0)	45	36	32
(20,0)	46	37	31
(21,0)	47	38	30
(22,0)	48	39	31
(23,0)	49	40	32

Fonte: execução do Protótipo 2, com o algoritmo de Custo Uniforme

Tabela 17 - Número de nós expandidos: Protótipo 2 - A*

Nó incial/ Nó final	(3,28)	(12,28)	(22,28)
(0,0)	34	41	51
(1,0)	33	40	50
(2,0)	32	39	49
(3,0)	33	38	48
(4,0)	34	37	47
(5,0)	35	36	46
(6,0)	36	35	45
(7,0)	35	34	44
(8,0)	36	33	43
(9,0)	37	34	42
(10,0)	38	33	41
(11,0)	39	32	40
(12,0)	40	31	39
(13,0)	39	30	38
(14,0)	40	31	37
(15,0)	41	32	36
(16,0)	42	33	35
(17,0)	43	34	34
(18,0)	44	35	33
(19,0)	45	36	32
(20,0)	46	37	31
(21,0)	47	38	30
(22,0)	48	39	31
(23,0)	49	40	32

Fonte: execução do Protótipo 2, com o algoritmo A*

APÊNDICE C: Uso de memória

Ilustração 47: Execução do algoritmo Busca em Largura 12x15 tiles - pico de uso de memória durante a execução

Ilustração 48: Execução do algoritmo Busca em Largura 24x29 tiles - pico de uso de memória durante a execução

Ilustração 49: Execução do algoritmo Busca de Custo Uniforme 12x15 tiles - pico de uso de memória durante a execução

Ilustração 50: Execução do algoritmo Busca de Custo Uniforme 24x29 tiles - pico de uso de memória durante a execução

Ilustração 51: Execução do algoritmo Best Search 12x15 tiles - pico de uso de memória durante a execução

Ilustração 52: Execução do algoritmo Best Search 24x29 tiles - pico de uso de memória durante a execução

Ilustração 53: Execução do algoritmo A* 12x15 tiles - pico de uso de memória durante a execução

Ilustração 54: Execução do algoritmo A* 24x29 tiles - pico de uso de memória durante a execução