IMPLEMENTAÇÃO INCREMENTAL NO ALGORITMO DE BUSCA DIJKSTRA: UMA ABORDAGEM APLICADA A JOGOS DIGITAIS[1]

LEONARDO MATHEUS TRENCH - leonardo15ma@hotmail.com

Centro Universitário de Araraquara - UNIARA

Resumo: Com a diversidade de aplicações em software possibilitadas pelas técnicas de Inteligência Artificial com destacada e ampla utilização delas em jogos digitais, este trabalho faz uma implementação incremental do algoritmo Dijkstra. O estudo está voltado à aplicação em jogos digitais. Jogos digitais e Inteligência artificial, devido a sua interconexão, formam a combinação necessária à sistematização em ciência da computação. Leva-se em consideração a necessidade de tornar os jogos mais interativos e com comportamentos mais próximos ao comportamento humano. Com um método de pesquisa de melhores práticas em ciência da computação e utilização de ferramentas conhecidas, como também a validação das técnicas incrementais em casos reais, os resultados obtidos a partir da implementação incremental do algoritmo Dijkstra em jogos digitais, contabilizam uma contribuição significativa para a área.

Palavras-chave: Algoritmo de busca, jogos digitais, inteligência artificial, Algoritmo Dijkstra.

Abstract: With the diversity of applications in software made possible by the artificial intelligence techniques with outstanding and extensive use of them in digital games, this work is an incremental implementation of Dijkstra Algorithm. The study is aimed at use in digital games. Digital games and artificial intelligence, due to its interconnection form necessary combination systematization in computer science. It takes into account the need to make games more interactive and closer to human behaviors. With a method of research of best practices in computer science and use of familiar tools, as well as the validation of incremental techniques in actual cases, the results obtained from the incremental implementation of Dijkstra algorithm in digital games, account for a significant contribution to the area.

Keywords: Search algorithm, digital games, artificial intelligence, Dijkstra algorithm.

1 INTRODUÇÃO

No contexto de jogos, é possível notar cada vez mais, a importância e influência que a Inteligência Artificial tem nos resultados no desenvolvimento de jogos.

[1] Trabalho de conclusão de curso, orientado pelo Prof. Dr. Heber Lombardi de Carvalho

A Inteligência Artificial consiste em buscar soluções para problemas complexos, tais como: ser capaz de reproduzir um comportamento humano (efetuar reconhecimentos de imagens, de objetos, de faces, de ações, de rotas e até sentimentos); construir e entender agentes inteligentes; construção de sistemas que tomem ação. Essas soluções são direcionadas ao jogador de modo à proporcionar conforto visual, escolhas, dificuldades técnicas para solução do jogo e caminhos complexos que resultam no processo de entretenimento. (SCHWAB, 2004).

Dentre as áreas da Inteligência Artificial, como: Redes Neurais, sistemas especialistas, lógica incerta, reconhecimento de voz, robótica, e outros, a técnica de busca ganha destaque por estar presente em aplicações distintas de softwares com diferentes propósitos. Conforme Dalmau (2004) o que se nota, é a variedade de algoritmos de busca e a incidência desses em jogos eletrônicos, portanto, este trabalho baseia-se na técnica de busca, onde na Inteligência Artificial, a “busca” se refere a uma técnica de procurar soluções para um problema (WHITBY, 2004).

Dentro do contexto dos algoritmos de busca direcionados para jogos, alguns jogos como *Leage of Legends* (RIOT), DOTA (Blizzard), Tibia (Cipsoft), *Smite* (Hi-Rez Studios) utilizam-se de algoritmo de busca. Os algoritmos de buscas também podem ser encontrados em gêneros de jogos como MOBA *(Multiplayer online battle arena), Shotter, Tower Defense* e RPG *(Role-playing game),* onde agentes, jogadores controlados pelo computador, munidos de Inteligência Artificial e mais especificamente da técnica de busca, definem a melhor rota para encontrar o destino que está buscando a partir de sua localização, independentemente da localização de onde o destino esteja.

Sendo assim, este estudo se baseia na implementação e modificação do algoritmo de Dijkstra, que consiste em um dos mais antigos já desenvolvidos, sistematizando uma nova execução para que em seu desempenho venha evitar e prever erros durante sua implementação e execução em jogos digitais.

Além da implementação de um algoritmo de busca, o trabalho se justifica em aplicar o algoritmo Dijkstra em jogos onde necessitam que agentes encontre rotas até determinados destinos, tendo como base resolver erros comuns como: evitar traçar caminhos errados, caminhos mais cumpridos ou mesmo não definir nenhum caminho possível. Com isso, efetuar melhorias buscando desempenho superior ao atual e propiciar melhores resultados durante sua execução.

Há jogos onde a Inteligência Artificial é aplicada, que durante o desenvolvimento deste trabalho, foi possível observar falhas em sua execução. Com a análise e alterações sistematizadas é possível implementar buscas mais eficientes que evitem as falhas comumente encontradas.

Conforme Lamothe (1999), Dalmau (2004) e Kishimoto (2004) a eficiência do algoritmo Dijkstra permite a utilização em diversas aplicações direcionadas para resolução de problemas em busca. O trabalho analisa e implementa melhorias para adequar e resolver o problema de pesquisa.

Como este estudo se trata da aplicação deste algoritmo melhorado em jogos, utilizou-se a ferramenta Unity, um ambiente para desenvolvimento de jogos, onde foi desenvolvido toda a parte de implementação e aplicação do algoritmo.

Após a escolha da ferramenta para o desenvolvimento e do algoritmo usado como base para todo este estudo, algoritmo Dijkstra, iniciou-se o processo de implementação, que consiste em passar a idéia do algoritmo para o código, implementando, de modo que possa ser executado em jogos digitais.

O funcionamento deste algoritmo consiste em pontos (*waypoints*) posicionados estrategicamente no mapa, de modo que um ponto ligado ao outro possa ter toda a visão do mapa, e assim, podendo guiar qualquer agente a qualquer destino.

O algoritmo Dijkstra, depende da utilização de *waypoints* para seu funcionamento, que são pontos posicionados no mapa, estrategicamente, responsáveis por mapear um determinado ambiente. (DALMAU, 2004).

Com a implementação sistematizada, é possível modificar e aperfeiçoar este algoritmo, de modo que ele se torne mais rápido e eficaz.

Sendo assim, o resultado do desenvolvimento e da implementação do algoritmo abordado pelo trabalho, permite a aplicação das melhorias em dois tipos de jogos específicos, sendo um jogo do gênero TPS *(Third-Person Shooter)* e outro *Tower Defense*, onde embasa os resultados.

2 REVISÃO BIBLIOGRÁFICA

A utilização da Inteligência Artificial (IA) em jogos e em *softwares* está cada vez mais sendo necessária para proporcionar reais melhorias em sua utilização (KISHIMOTO, 2004). A aplicação de IA tem por objetivo recriar a inteligência

humana e desenvolver suas técnicas de modo a criar *softwares* e agentes inteligentes (SCHWAB, 2004, MICHALEWICZ, 2004, FOGEL, 1999), deste modo, a IA preocupa-se em como o sistema age e não como ele pensa, sendo assim, seu foco e importância está voltado no resultado, a fim de proporcionar diversão e jogabilidade para o jogador (TOZOUR, 2002, HOLLAND, 1993, BÄCK, 1997).

Dentro do contexto da Inteligência Artificial, existem inúmeros algoritmos como demonstra Whitby (2004) com a utilização de algoritmos de buscas. O algoritmo de busca consiste em uma técnica onde se traça uma rota de um ponto de origem a um destino, e sem ter o conhecimento do caminho a ser percorrido entre o ponto de origem e o ponto de destino, seja capaz traçar uma ótima rota para poder alcançá-lo (MITCHELL, 1996).

Para Dalmau (2004), a utilização de técnicas e algoritmos em jogos digitais é imprescindível. O autor também evidencia a utilização de algoritmos de busca como o Algoritmo Dijkstra. Segundo Lamothe (1999), a utilização do algoritmo Dijkstra vai além da aplicação estar restrita a jogos.

O Dijkstra é um algoritmo de busca que se utiliza de pontos devidamente posicionados proporcionando visibilidade do mapa quando interligados. Isso significa que os pontos conectados formam uma espécie de teia dentro do mapa, fornecendo a visão total, para então, através da ligação entre os pontos, ser capaz de traçar uma rota de um ponto de origem até um ponto de destino sem mesmo ter tido um mapa como *input* previamente.

3 MÉTODO E DESENVOLVIMENTO

3.1 Método

O método utilizado neste estudo, contempla os assuntos de IA e, a implementação e otimização de uma de suas técnicas em jogos digitais. Vinculado a estes assuntos, levando em conta que IA e jogos digitais se complementam, busca-se conceitos sobre a aplicação dos mesmos e a implementação da técnica de IA, de busca, em jogos digitais.

Desta forma, adequando-se aos conceitos de Waslawick (2009) e Wainer (2007), é possível citar os passos seguidos para a realização do desenvolvimento deste estudo.

Os passos são: Procedimentos operacionais, estudo e levantamento de possíveis falhas do algoritmo Dijkstra, aplicação e utilização da ferramenta Unity, implementação do estudo incremental, detecção e correção de erros, testes e avaliação de desempenho e por fim a otimização.

3.2 Procedimentos operacionais

Como primeiro passo, é notado um problema para traçar rotas a destinos com rotas desconhecidas, no qual poderia ser solucionado com o algoritmo de busca. Então, escolhe-se o algoritmo Dijkstra. No segundo passo, é escolhido uma ferramenta que possibilite a exemplificação do estudo e a aplicação do algoritmo, portanto, escolhido a ferramenta Unity. Terceiro passo, é realizado a implementação básica do algoritmo Dijkstra simulado em um cenário de jogo digital. No quarto passo, é simulado e forçado a apresentação de erros. No quinto passo, é analisada e realizada uma avaliação do desempenho da implementação. No sétimo passo, é realizado a otimização do algoritmo. No oitavo passo, é feita a adequação do algoritmo para destinos móveis (que se movimentam). E por fim, o último passo é a aplicação do resultado dos passos anteriores, demonstrados em dois jogos.

3.3 Algoritmo Dijkstra

O algoritmo de busca Dijkstra foi desenvolvido em 1956 e teve sua publicação oficial em 1959. Sua primeira utilização foi em aparelhos GPS *(Global Positioning System),* (BARROS et. al., 2007).

Este algoritmo consiste em pontos estrategicamente posicionados de modo que, através destes, seja possível chegar até um destino pelo menor caminho possível, considerando a distância entre eles e, para isso, em cada ponto é atribuído um valor para que seja realizado este cálculo.

É selecionado um ponto de origem, juntamente com outros pontos interligados a ele, e tendo todos seus valores atribuídos, é possível que através de cálculos seja definido a rota mais curta para chegar ao seu destino, ou seja, de um ponto X a um ponto Y.

Para a exemplificação da funcionalidade deste algoritmo, é demonstrado um pseudocódigo do algoritmo *DIjkstra* como mostra a Figura 1.

Figura 1 – Algoritmo Dijkstra

```
function Dijkstra(Graph, source):
for each vertex v in Graph:                 // Initialization
    dist[v] := infinity                     // initial distance from source to vertex v is set to infinite
    previous[v] := undefined                // Previous node in optimal path from source
dist[source] := 0                           // Distance from source to source
Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
while Q is not empty:                       // main loop
    u := node in Q with smallest dist[ ]
    remove u from Q
    for each neighbor v of u:               // where v has not yet been removed from Q.
        alt := dist[u] + dist_between(u, v)
        if alt < dist[v]                    // Relax (u,v)
            dist[v] := alt
            previous[v] := u
return previous[ ]
```

Fonte: TARAU (2013).

3.4 Unity

O Unity é uma ferramenta para desenvolvimento de jogos intuitiva e simples, e por meio dela, é possível atingir um nível de desenvolvimento que cumpre os requisitos mínimos em games. Além disso, possui suporte ao usuário e é gratuita.

O Unity utiliza como base linguagens como, Javascript, Boo e *C#.* Sendo assim, para a execução deste trabalho utiliza-se a linguagem *C#* pela robustez, estabilidade e difusão que garantem a implementação do algoritmo.

3.5 Implementação

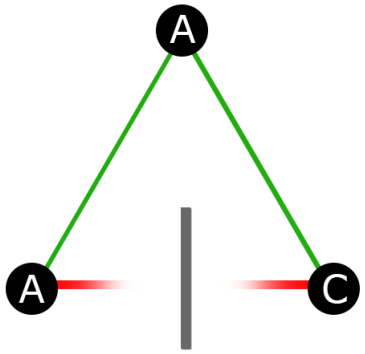
Assim que definido a utilização da ferramenta Unity como plataforma-base de desenvolvimento somada a estruturação fornecida pelo algoritmo de busca Dijkstra inicia-se a fase de implementação.

Para que a implementação se viabilize, cria-se o cenário de um jogo para a simulação do algoritmo. E, então, define-se um agente. Um personagem utilizado para simular a execução do algoritmo, onde ele deve percorrer os pontos que são as coordenadas referenciadas no mapa do jogo. Logo após, adiciona-se os pontos, de

forma que se referenciem e com isso tenha a visão do mapa, onde um ponto tenha visão do outro e deste modo o mapa inteiro fica referenciado entre os pontos, numa espécie de teia de aranha. Como os mapas de jogos sempre apresentam algum tipo de obstáculo (parede, pedras, algo que impeça a passagem), os pontos só podem se referenciar quando não há obstáculos entre si, desta forma, não é traçada rotas que o agente não consiga passar, pois o obstáculo está em seu caminho.

Para exemplificar como os obstáculos influenciam nas referências entre os pontos, imagine um cenário onde temos três pontos A, B e C e uma parede. Neste cenário, parede se localiza entre os pontos B e C, desta forma, não se enxergam, e não podem se referenciar. O ponto A pelo contrário, enxerga os pontos B e C, ou seja, é estabelecido uma referencia entre os pontos A-B e A-C já que não há obstáculo entre eles, como mostra a Figura 2. Então, só pode ser traçada a rota entre pontos que se referenciem, pois assim não terá obstáculos que impeçam a passagem do agente, podendo chegar ao destino estabelecido.

Figura 2 – Demonstração da explicação de obstáculos



Fonte: Elaborado pelos autores.

Para início da implementação, utiliza-se um agente, dez pontos e cinco paredes.

As paredes são posicionadas de modo aleatório e então, são adicionados os pontos de forma estratégica, ou seja, de modo que formem uma teia entre si, estabelecendo referencias entre eles, considerando os obstáculos. É importante deixar claro que pontos sem referências, isolados, não fornecem ao algoritmo nenhuma alternativa para que a rota seja traçada, já que a rota é o caminho considerando os pontos como passos a serem seguidos, por isso, a importância de posicionar estrategicamente os pontos. O agente é criado e nele é atribuído a tarefa de

visualizar o ponto mais próximo de sua localização e iniciar a rota a partir deste ponto. Para que se inicie a rota, o agente informa qual o destino que deseja alcançar, e então, com as referências já estabelecidas entre os pontos, é possível traçar a melhor e mais curta rota, considerando a distância entre os pontos até o destino informado, já que todos se enxergam mesmo que indiretamente. Desta forma é estabelecida a rota que o agente poderá percorrer seguindo os pontos até seu destino.

3.6 Correção de Erros durante a execução

Com a implementação passa-se a identificar os erros que ocorrem durante a execução do jogo, de forma aleatória ou com a intervenção humana.

Durante a execução do jogo, ocorre que, o agente percorre do ponto A até o ponto B e volta ao ponto A, devido à menor distância entre eles, e, desta forma ele não sai destes pontos, não alcançando rota alguma e nem traçando/percorrendo a rota esperada. Para a resolução deste problema, é atribuída uma variável que grava o último ponto percorrido, o ponto A, para que não voltasse ao mesmo ponto e então evitasse que o destino nunca fosse alcançado, desta forma, após o agente percorrer do ponto A ao ponto B, ele, o agente é impedido de voltar ao ponto A, sendo necessário que o agente siga para o próximo ponto, diferente do anterior.

Outro erro que previamente se observa, é quando o destino informado não é visível de nenhum ponto, deste modo, não é possível que entre as referências dos pontos, seja traçado a rota ao destino, e assim, o agente anda em círculos ou então não toma nenhuma ação (não se movimenta). Em ambos os casos, o agente não chega ao destino esperado, por isso, a importância do posicionamento correto dos pontos cobrindo todo o mapa e estabelecendo uma visão total dele, para que não tenha pontos cegos e inalcançáveis.

3.7 Avaliação de desempenho

O desempenho com um agente significa quando ele alcança rapidamente e sem surpresas ou situações inesperadas o destino desejado.

Porém em jogos que utilizam algoritmos de busca, a definição de rotas com apenas um agente é rara, já que não existe apenas um agente simultâneo no jogo,

então, distribuem-se no mapa mais agentes com o mesmo algoritmo para que seja possível analisar o desempenho da execução com mais agentes.

Para a avaliação do desempenho, utiliza-se a técnica de testes de carga, que é baseada em aumentar a carga do processamento e verificar o quanto ela exerce influência durante a execução do jogo, buscando um limite ou alguma mudança.

Em outros jogos, como *Leage of Legends* e Dota, os respectivos mapas são maiores, e possuem maior número de obstáculos e agentes, percebe-se então, a necessidade de aumentar a quantidade de obstáculos, de cinco para dez. Sendo assim, com mais obstáculos, altera-se o tamanho do mapa e a quantidade de pontos de dez para trinta e dois, para que os pontos consigam ter a visão total do mapa. Atribui-se então dez, vinte e trinta agentes no mesmo cenário.

Ao modificar a quantidade de pontos no mapa, seja duplicando ou aumentando cinco pontos, é possível notar um aumento no custo de processamento, já que o algoritmo faz uma leitura ponto por ponto para cada agente. Então, um ponto a mais, significa que cada agente sofra um atraso individual considerável no processamento. O processamento completo do jogo, analisando o conjunto e não cada agente separadamente, a velocidade de processamento de 65 FPS (frames por segundo, que consiste na quantidade de quadros construídos em um segundo) passou para a faixa dos 15 FPS aos 35 FPS, na execução do algoritmo, considerado o pior caso apresentado na execução com os 30 agentes, onde nesta quantidade de FPS, a execução do jogo apresenta travamentos, ou seja, lentidão no processamento.

Abaixo, a Quadro 1 demonstra a alteração de FPS com o aumento de agentes e dos pontos:

Quadro 1 – Demonstração de desempenho

Agentes	Pontos	FPS (tempo)
05	10	60-75
10	32	21-45
20	32	17-39
30	32	15-35

Fonte: Elaborado pelos autores.

3.8 Otimização do código

Demonstrada, pelos testes executados, então, que a quantidade de pontos é o fator principal para o aparecimento de atrasos durante a execução. Faz-se assim, necessárias as alterações na implementação original do algoritmo Dijkstra.

Sendo elas: o reaproveitamento de rotas já traçadas, fazendo com que não necessite processar novamente sempre as mesmas informações; A adaptação do algoritmo para quando o destino é fixo, não sofrendo mudança e portanto precisando apenas de um processamento inicial para traçar a rota;

Como os jogos são executados sobre uma execução sem fim, como um laço repetitivo (*loop*), é preciso então que seja liberado esse processamento quando ele começa a apresentar gargalos, ou seja, necessário atribuir pontos para a liberação de processamento.

Os pontos de liberação do processamento, consiste em locais no código, onde o processamento é liberado para o frame ser construído. Os jogos são executados em loop, e um frame consiste no termino de um ciclo executado pelo loop principal do jogo, e portanto, para que um jogo seja dinâmico, vários frames por segundo (FPS) são construídos. Imaginando um cenário onde um personagem anda pela tela, dentro dessa movimentação, considera-se que tenha sido executados movimentos, como o movimento de andar, o movimentar de membros do corpo representa a construção de um frame, então, vários frames consecutivos mostra uma sequência imperceptível de movimentos, como um vídeo/animação. Considerando que se torne infinito ou lento a finalização de um ciclo, o frame não é construído, causando travamento ou lentidão do jogo. Como uma solução para este problema, é necessário diminuir a atividade de processamento da máquina, inserindo pontos de liberação, com a seguinte condição de parada: 1) variável de marcação de parada no processamento é chamada para indicação da localização do encerramento parcial do ciclo; 2) retomada da elaboração do "frame" 3) após a finalização do frame retoma-se o processamento a partir da variável de indicação de parada.

Colocando esses pontos de liberação de processamento em locais inadequados no código, como no início de funções ou tarefas que são rápidas, pode ocasionar um retardamento do processamento em um frame, considerando que o sistema é forçado a diminuir a atividade do processamento. Portanto, é necessário tomar os cuidados para construção de um código com baixo desempenho, evitando

laços de repetição, evitando funções recursivas e entender como realmente o jogo funciona e se há necessidade da inserção de pontos de liberação de processamento.

Com a necessidade de liberar o processamento durante a execução, para evitar que o algoritmo, quando em operação trave, analisa e define-se. Conforme o alto custo de processamento e as funções executadas por vários agentes simultaneamente, quanto maior a quantidade de agentes, maior o custo para a execução destes funções. Então, por meio de teste de carga, alterando o posicionamento dos pontos de liberação e alterando a quantidade de agentes, analisando a alteração no número de FPS durante a execução do jogo, são definidos os locais eleitos para as inserções dos pontos de liberação de processamento no código.

3.9 Resultado da otimização

Com a otimização concluída, realiza-se o teste de aumentar agentes, obstáculos e consequentemente os pontos presentes no cenário, para comprovar as mudanças conquistadas.

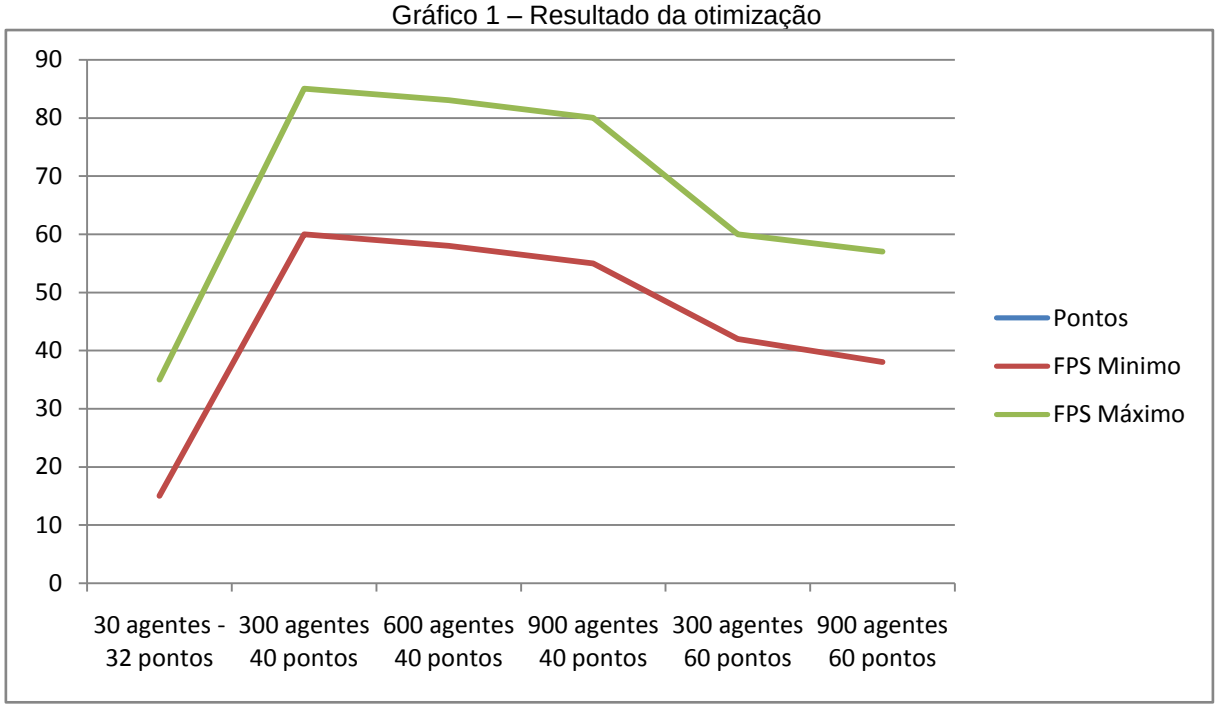
Com a inserção de 300 agentes e quarenta pontos em um cenário, realiza-se o teste onde mostra que com quarenta pontos posicionados, a simulação é executada na faixa dos 60 FPS a 85 FPS, porém, acima dos sessenta pontos, o risco de atrasos aumenta a uma taxa de trinta por cento (30%) durante a execução, já que, quanto mais pontos, maior o custo de processamento para cada agente. Já os agentes, ao aumentar sua quantidade, de 300 para 600 e depois 900, não houve alterações significativas em desempenho como mostram a Quadro 2 e o Gráfico 1, comparando o FPS durante o jogo.

Quadro 2 – Resultado da otimização

Agentes	Pontos	FPS (tempo mínimo - máximo)
300	40	60-85
600	40	58-83
900	40	55-80
300	60	42-60
900	60	38-57

Fonte: Elaborado pelos autores.

Gráfico 1 – Resultado da otimização



Fonte: Elaborado pelos autores.

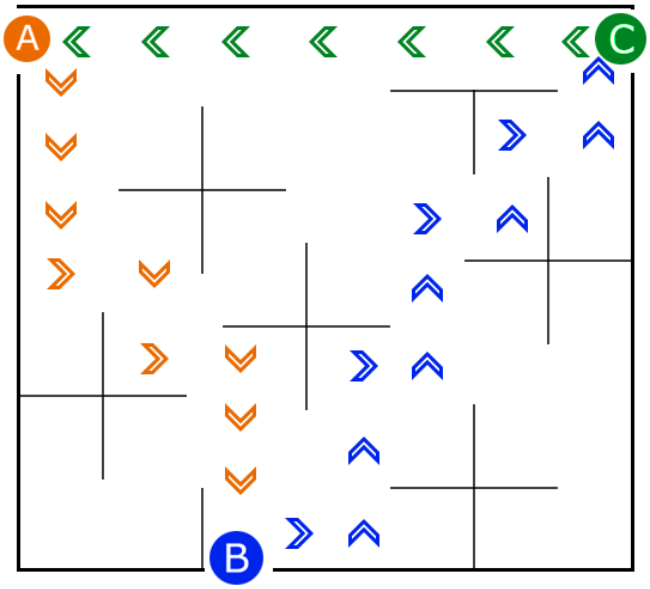
3.10 Adaptação para destinos móveis

A adaptação que se faz premente é para que o algoritmo seja capaz de traçar sua rota além de destinos onde não há variação de posição, fixos, mas também para destinos que se movam, bem como seguir e definir rota até outros agentes, onde ambos estão em movimento.

A adaptação a ser feita no algoritmo, se refere a redefinir a rota constantemente, proporcionando sempre a proximidade referente a possível nova posição do destino que deseja alcançar, esteja ele parado ou que tenha se movido.

Um exemplo. Considerando um labirinto com três entradas, onde em cada uma delas está presente um agente, sendo assim, o destino do agente A é o agente B, o destino do agente B é o agente C e o destino do agente C é o agente A, e desta forma, as rotas são traçadas até o ponto de origem de seus respectivos destinos, conforme mostra a Figura 3.

Figura 3 – Demonstração de destinos móveis



Fonte: Elaborado pelos autores.

Considerando que as rotas definidas, utilizam como base, a posição inicial do destino, ou seja, quando a rota foi traçada pela primeira vez, se o agente permanecer com a mesma rota, porém, seu destino se moveu ou sua posição mudou, isso representa que o destino não será alcançado, já que a posição da rota já traçada é a posição inicial do destino.

Portanto, é necessário que a rota seja recalculada sempre que o destino sofrer mudanças em sua posição, considerando um tempo para que seja recalculada, como no caso, de quando o agente alcança o próximo ponto e verifica que seu destino sofreu mudanças na posição. E solicitado então, que sua rota até seu destino seja recalculada e desta forma, mesmo com o destino em movimento, o agente é capaz de alcançá-lo.

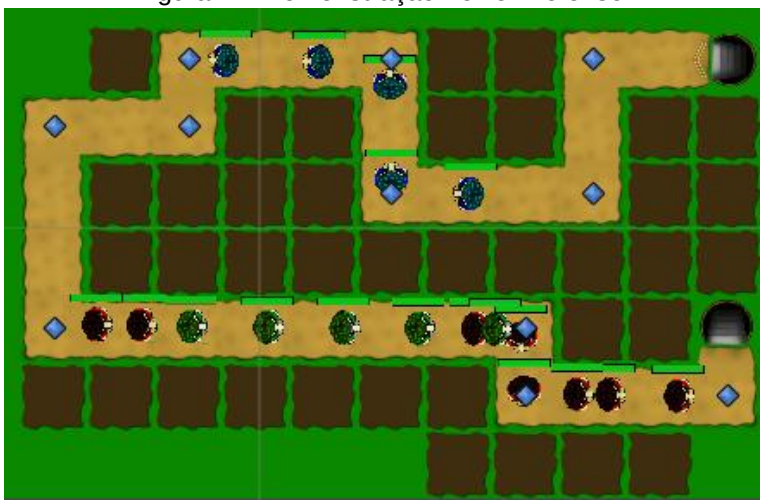
3.11 Aplicação em cenários diferentes

Para validar as funcionalidades dos resultados alcançados e demonstrá-los, o algoritmo implementado e modificado até aqui é aplicado em dois gêneros de jogos, aplicando os dois conceitos de destinos: móveis e fixos.

Como primeira demonstração, ilustrada pela Figura 4, desenvolve-se um jogo do gênero *Tower Defense* (TD), onde o destino é fixo, já que consiste dos agentes cruzarem um mapa até um destino. No TD, que consiste em um jogo onde o objetivo

é impedir a chegada de agentes até um determinado destino, os agentes para tal são criados em forma de ondas. Então, são criados 10 agentes por onda e, portanto, a cada nova onda, realizam-se novos cálculos para a definição da rota até o destino.

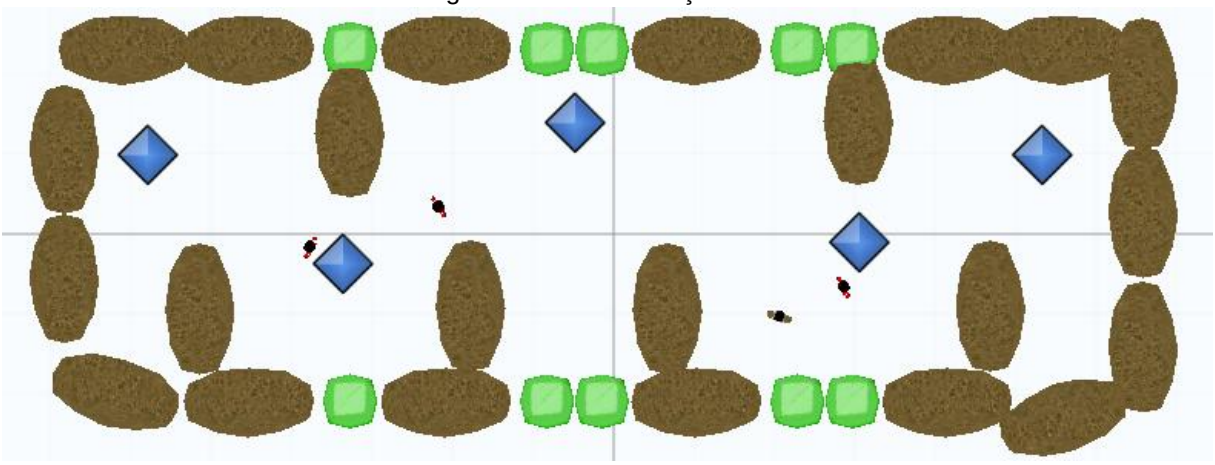
Figura 4 – Demonstração Tower Defense



Fonte: Elaborado pelos autores.

Como segunda demonstração, como ilustra a Figura 5, desenvolve-se um jogo do gênero TPS (*Third-Person Shotter*) de modo que um personagem seja seguido por agentes e mesmo que se movimente, esses agentes o encontrem. No TPS, existem dois locais no mapa onde são responsáveis pela criação dos agentes, desta forma, os agentes ao serem criados, já são guiados até a posição do personagem, e, com o conceito de destino móvel, conforme o personagem se movimenta, muda de posição, o agente alcança o próximo ponto, onde sua rota é recalculada.

Figura 5 – Demonstração TPS



Fonte: Elaborado pelos autores.

Ambas as demonstrações, contam com obstáculos e agentes sendo criados aleatoriamente, deste modo, validando a execução do algoritmo aplicado, com destino atribuído fixo e móvel.

4 RESULTADO

O trabalho resolve três cenários computacionais. Primeiro, a correção e a prevenção dos erros encontrados em aplicações que utilizam algoritmo de busca, tais como, não alcançar o destino, erro ao definir/calcular a rota, ou traçar uma rota que seja a mais longa. Segundo, o melhor desempenho do algoritmo desenvolvido em relação à implementação inicial, ocorrida na análise de consumo de processamento e observando os travamentos e lentidão durante a execução do jogo, conforme detectada pela pesquisa inicialmente, que possibilita o aumento de mais de 1000% (antes trinta agentes e depois mais de 300) no número de agentes e mais de 400% (antes dez pontos e depois mais de quarenta) no numero de pontos. Terceiro, a adaptação do algoritmo que possibilita a rota já traçada, ser recalculada ou refeita quando o alvo for definido como móvel, ou seja, quando ele sofre alteração de posição.

5 CONCLUSÃO

A contribuição é a otimização do algoritmo de busca Dijkstra em ambientes/cenário para jogos, com ganhos demonstrados tanto para evitar quanto para tratar erros, como: travamentos inesperados durante a execução; rotas que são traçadas/definidas incorretamente; rotas cuja distancia não é a menor possível; rotas definidas sem que o destino tenha sido, de fato, encontrado; a não capacidade de definir rotas.

Para a otimização, além dos erros evitados e tratados, há também, como resultado, a obtenção da realização do trabalho, o aumento do desempenho na execução do algoritmo, independentemente do status do destino informado ser móvel (com movimentação) ou fixo (sem se movimentar).

Isso possibilita agregar ao resultado do estudo a característica de flexibilidade. Essa característica permite a adaptação do algoritmo em jogos de diversos gêneros. Um exemplo são os que necessitam traçar rotas para destinos desconhecidos, o que, demonstrado pelo trabalho, leva a utilização e aplicação de um algoritmo de busca.

Desenvolvido este estudo e, a implementação e adequação do algoritmo de busca Dijkstra em jogos, fica claro o quão presente está a Inteligência Artificial. Assim, as maquinas reproduzem e aplicam o raciocínio humano por intermédio da captura de conhecimento de especialistas, o que faz das máquinas, em certo aspecto, um gênero de inteligência a serviço e bem do ser humano.

O estudo enfatiza a importância de códigos otimizados de modo a influenciar de maneira positiva durante a execução do jogo, como comprovado pelo trabalho.

A implementação do algoritmo Dijkstra neste estudo, demonstra também, que há casos onde, os algoritmos ao serem implementados em diferentes ambientes, necessitam ser incrementados e adequados.

Desta forma, após a apresentação de redução de custos de processamento através da otimização, seguida da apresentação do conceito de algoritmo de buscas em Inteligência Artificial, dos passos realizados para a implementação do algoritmo e dos desafios durante o desenvolvimento como apresentado neste artigo, conclui-se que este estudo fornece comprovações. Tais como algoritmos conceituados, em diferentes ambientes de implementação passam por mudanças e devem ser adaptados e incrementados sistematicamente. O que essencialmente leva a otimização de código para o aumento do desempenho das aplicações.

A otimização obtida, bem como a demonstração de sua eficácia, remetem à uma expansão para a área. E, sugere que trabalhos e aplicações distintas podem ser beneficiados com a utilização das técnicas desse trabalho aumentando a probabilidade de sucesso nas adaptações incrementais sistematizadas.

REFERÊNCIAS BIBLIOGRÁFICAS

BÄCK, T., FOGEL, D.B. & MICHALEWICZ, Z. (eds.) “**Handbook of Evolutionary Computation**”, Institute of Physics Publishing and Oxford University Press, 1997.

BARROS E. A. R., Pamboukian S. V. D., et. al., **Algoritmo de Dijkstra: Apoio Didático e Multidisciplinar na Implementação, Simulação e Utilização Computacional,** International Conference on Engineering and Computer Education, 2007.

DALMAU, Daniel Sánchez-Crespo. **Core Techniques and Algorithms in Game Programming.** Indianopolis: New Riders. 2004.

FOGEL, D.B. "**Evolutionary Computation: Toward a New Philosophy of Machine Intelligence**", 2nd edition, The IEEE Press, 1999.

HOLLAND, J.H. **Induction : processes of inference, learning, and discovery**, Cambridge : The Mit Press, 1993.

KISHIMOTO, André. **Inteligência artificial em jogos eletrônicos.** 2004. Disponível em: http://www.karenreis.com.br/pdf/andre_kishimoto.pdf. Acesso em: 17 junho 2015.

LAMOTHE, André. **Tricks of the Windows Game Programming Gurus – Fundamentals of 2D And 3D Game Programming.**

MICHALEWICZ, Z. **Genetic algorithms + Data Structures = Evolution Programs**, 3rd edition, Springer-Verlag, 1994.

MICHALEWICZ, Z.; FOGEL, D. B. **How to solve it: Modern Heuristics**, Springer-Verlag, 2004.

MITCHELL, M. **An Introduction to Genetic Algorithms**, The MIT Press, 1996.

SCHWAB, Brian. **AI Game Engine Programming.** Hingham: Charles River Media. 2004.

TARAU, Paul. **Dijkstra's Algorithm**. 2013. Disponível em:

http://www.cse.unt.edu/~tarau/teaching/AnAlgo/Dijkstra's%20algorithm.pdf. Acesso em: 27 outubro 2015.

TOZOUR, Paul. **The Evolution of Game AI from AI Game Programming Wisdom.** Hingham: Charles River Media. 2002.

WAINER, J. **Métodos de pesquisa quantitativa e qualitativa para a ciência computação**. In T. KOWALTOWSKI e K. BREITMAN (Org.), Atualização em Informática da Sociedade Brasileira de Computação. 2007.

WASLAWICK, R.S. **Metodologia de Pesquisa para Ciência de Computação**. Elsevier Editora Ltda. 2009.

WHITBY, BLAY. I.A. **Inteligência Artificial: um guia para iniciantes.** São Paulo: Editora.