Lucas Teles Agostinho
Rodrigo Mendonça da Paixão

Estudo de algoritmos genéticos para busca de caminhos.

São Paulo – Brasil

2016

Lucas Teles Agostinho
Rodrigo Mendonça da Paixão

Estudo de algoritmos genéticos para busca de caminhos.

Pré-monografia apresentada na disciplina Trabalho de Conclusão de Curso I, como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação.

Centro Universitário Senac

Bacharelado em Ciência da Computação

Orientador: Eduardo Heredia

São Paulo – Brasil

2016

Lista de abreviaturas e siglas

AG	Algoritmos Genéticos
A*	AStar ou A Estrela
IA	Inteligência Artificial
CPU	Central Processing Unit]
PCV	Problema do Caixeiro Viajante
BFS	Best-first search
PPGA	Pattern based Pathfinding with Genetic Algorithm

Sumário

1 Introdução

1.1 Motivação

O problema de buscar o melhor caminho entre dois pontos tem uma grande importância em áreas como da engenharia e ciência, tais como rotear o tráfico de telefone, navegar por um labirinto ou mesmo definir o layout de trilhas impressas em uma placa eletrônica.

Busca de caminhos também tem uma grande importância no âmbito dos jogos digitais, onde um jogador compete ou coopera com uma inteligência artificial e é preciso chegar ao seu destino de forma competente, como por exemplo, jogos de tiro em primeira pessoa ou de estratégia em tempo real.

O valor de entretenimento do jogo pode ser drasticamente reduzido, quando os personagens não podem atravessar um mapa complexo de forma competente, podendo afetar a experiência de jogo ao deixar visível para o jogador a sua incapacidade de lidar com a busca de caminho de forma satisfatória.

Ainda é comum, em jogos digitais, termos mais de um agente de busca de caminho ao mesmo tempo no mesmo cenário, podendo ser muitas vezes muito custoso computacionalmente falando. Para isso vários desenvolvedores de jogos têm juntado esforços para desenvolver soluções de busca de caminho em ambientes de recursos escassos.(PONTEVIA, 2008)

Muitas abordagens para melhorar o desempenho ou diminuir o custo de métodos de busca de caminho tem sido desenvolvidos (SANTOS, 2012a) (POLLACK, 1960) (CAIN, 2002) (MILLER, 2010).

Ainda temos problemas de alto custo computacional, que para serem minimizados em alguns casos, acabamos sacrificando a certeza de melhor caminho por um melhor desempenho(BOTEA, 2004)(KORF, 1993)(RUSSELL; NORVIG, 2003). Por isso iremos faremos uma análise e propor um modelo que visa otimizar o seu consumo de memória, visando em principal o âmbito dos jogos digitais.

Qualquer problema que possa ser formulado em busca em grafos pode ser solucionado a partir de busca heurísticas. O exemplo mais comum que pode ser representado por um grafo é o PCV, outro exemplo típico é quebra-cabeça de deslizar, que consiste em um quadro que contem quadrados numerados e uma posição vazia (Conforme a Figura 2). As operações permitidas são deslizar qualquer peça horizontalmente ou verticalmente adjacente a peça vazia para a posição vazia e o objetivo é rearranjar as peças a partir de

uma configuração randômica com menor quantidade de movimentos. Encontrar a solução ótima para esse problema ou para o PCF é NP-Completo. (KARP, 1972) (RATNER; WARMUTH, 1986)

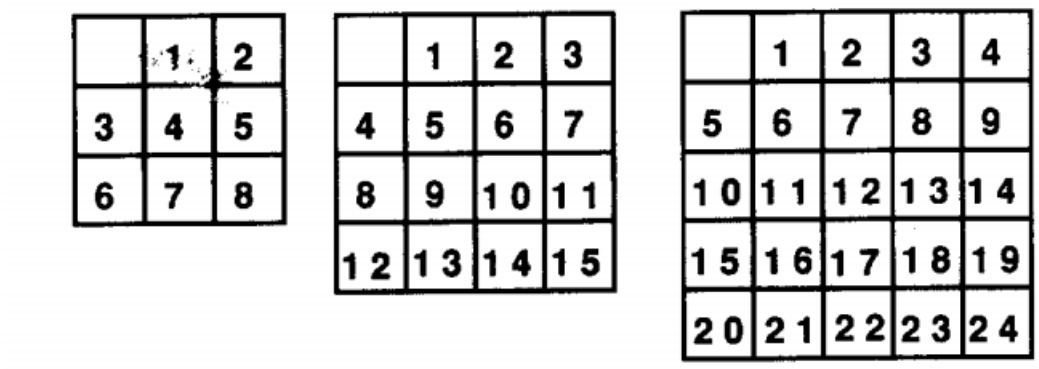

Figura 1 – Quebra cabeça de deslizar (KORF, 1993)

Este cenário é a principal motivação deste trabalho que consiste implementar e mensurar resultados de uma solução para busca de melhor caminho entre dois pontos.

1.2 Objetivos

Este trabalho tem como objetivo otimizar o consumo de memória exigido pela busca de caminhos combinando AG com a heuristica do algoritimo de busca. O ponto central é a definição de uma estrutura capaz de explorar as vantagens demonstradas no uso de AG com Busca de caminhos (BURCHARDT; SALOMON, 2006)(SANTOS, 2012b) aplicá-las em um modelo para o otimizar o custo de memória.

1.2.1 Objetivos Específicos

Implementar o modelo de busca heurística com AG PPGA e mensurar os resultados.

Verificar o desempenho em mapas com padrão e sem padrão.

Alterar os padroes de aprendizado e verificar como o algoritimo se comporta.

Propor uma alteração no modelo para tentar otimizar o custo de memoria.

1.3 Método de trabalho

Primeiramente será desenvolvido uma bateria de testes de caminhos, com intuito de terem diferentes tamanhos, levando em consideração mapas com padrões de repetição e sem padrões de repetição, os mesmos serão modelados de forma bidimensional em arquivos

de texto com caracteres para definir o ponto inicial (S), final (E), obstáculos (#) e caminho livre (.).

Será implementada a busca heurística A* e BFS utilizando a linguagem C# e .NET Framework da Microsoft, levando em consideração os arquivos de teste desenvolvidos como entrada, retornando se existe um caminho entre o ponto inicial e final, quanto tempo levou para encontrar o caminho e consumo de memoria para chegar na solução.

Também será implementado um AG para utilizar em conjunto ao A*, também utilizando a linguagem C# .NET, implementando o mesmo método proposto por Ulysses O. Santos (SANTOS, 2012a) o PPGA. A implementação receberá arquivos de testes de entrada contendo os mapas, os mesmos serão utilizado para testar a implementação do A* anteriormente, também retornando o tempo e consumo de memoria para chegar na solução. Os parâmetros para serem utilizados no AG serão:

Uma população inicial de 4 indivíduos

Critério de geração de descendentes em 50%

Critério de mutação em 25%

Aptidão para 12 indivíduos

Por último, será implementado uma modificação do modulo de AG. Serão utilizadas N *threads* para a separação das populações, ou seja, trabalhando com várias populações ao mesmo tempo, onde cada *thread* fica responsável por uma, para depois mesclar os melhores indivíduos entre elas, esse sistema ira receber os mesmo arquivos de testes utilizados nas implementações anteriores, retornando informações de tempo e consumo de memória.

Será calcula a média de tempo e consumo de memória entre cada um dos testes. Todos os dados coletados serão comparados em forma de tabelas e gráficos, para demostrar os ganhos e perdas relativos de cada abordagem.

1.4 Organização do trabalho

Este trabalho é dividido em 4 capítulos. O primeiro capitulo faz uma introdução geral do problema, descrever os objetivos e a motivação para a resolução do problema proposto.

O segundo capitulo trata do problema de forma separada, mostrando o que existe na literatura para uma possível solução. Também explica de forma mais detalhada o funcionamento de dois exemplos de busca heurística, demostrando uma aplicação em um trabalho da literatura e dos algoritmos genéticos, explicando seu funcionamento e aplicação na literatura. O terceiro capitulo é a proposta apresentada para a criação deste trabalho.

2 Revisão de Literatura

Nesse capítulo é feita uma revisão no estado da arte de diferentes formas de utilizar os algoritmos de busca, algoritmos genéticos, paralelismo ou uma união de todas ao mesmo tempo.

2.1 Busca Heurística

Busca heurística de caminhos é um subgrupo de algoritmos de busca em grafos (WINSTON, 1992), é um mecanismo geral de solução de problemas no ramo de IA. A sequência de passos necessários para solução da maioria dos problemas de IA usualmente não é conhecida, mas deve ser determinada por uma exploração de alternativas através de mecanismos de tentativa e erro. Um problema de busca típico é o PCV, que consiste em encontrar o caminho mais curto que as visite cada cidade de um conjunto cidades e retorne para a cidade de partida. Os algoritmos BFS e o A* são otimo exemplos de busca heurística. (KORF, 1993)

2.1.1 *Best-first search*

É um algoritmo de busca heurística geral. Ele mantém uma lista aberta contendo os nós ainda não visitados de uma árvore que é gerada, e uma lista fechada dos nós que já foram expandidos e visitados. Cada nó tem um valor de custo, em cada ciclo um nó de menor custo da lista aberta é expandido, cada filho do nó é avaliado por uma função de custo 2.1 e adicionado na lista aberta, e o nó pai é adicionado a lista de nos fechados.

$$f(n) = h(n) \tag{2.1}$$

A lista aberta contem apenas o nó inicial e o algorítimo termina quando encontrar o nó de destino para expansão.(KORF, 1993)

Existem muitas implementações com alterações do BFS, tais como *Dijkstra* ou A*, o que os diferencia basicamente é como é implementada sua função de custo.

2.1.2 O Algoritmo A*

O algoritmo A* é um dos mais populares soluções no ramo de busca de caminhos, um desses motivos é que sempre garantindo achar o menor caminho entre dois pontos (HART; RAPHAEL, 1968), porém, gera uma grande árvore de busca nos processos, consumindo muito tempo e memória. É comum haver modificações no algoritmo para explorar uma

árvore de busca menor, diminuindo o tempo para achar um caminho sacrificando a garantia de se encontrar o melhor caminho no final. (BOTEA, 2004). O algoritmo A*, em sua forma tradicional, utiliza a fórmula heurística 2.2.

$$f(n) = g(n) + h(n) \tag{2.2}$$

Onde g(n) é o custo para chegar ao nó n, e h(n) é o custo estimado para atingir o nó de destino a partir do nó n. Este sempre deve ser menor ou igual a distância real entre o ponto n e o destino, ou seja nunca pode ser super-estimado. Para cada iteração sobre os vizinhos do nó atual é calculado o f(n) e adicionado em uma lista de nós abertos (A), mantendo uma referência do nó que serviu de origem para chegar nele, caso o nó ja esteja na lista e o valor de f(n) novo for menor, o valor de f(n) e o nó de origem é substituído pelo novo. Depois é verificado na lista o menor valor de f(n), este é removido, adicionado a lista de nos fechados (F) e a partir desse ponto, ele se torna o nó atual, quando o nó atual é o mesmo nó de destino o algorítimo retorna o caminho encontrado, assim podemos encontrar a solução ideal. (HART; RAPHAEL, 1968)

Podemos aplicar o algoritmo A* em um Grafo direcionado ponderado por exemplo da Figura 2.

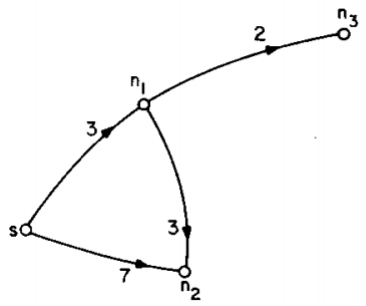

Figura 2 – Grafo para busca (HART; RAPHAEL, 1968)

O grafo consiste em um nó inicial s e mais três outros nós (n1, n2, n3). As arestas contem a direção e o custo do trajeto. Se partirmos do algoritmo A* para produzir um subgrafo do melhor caminho, partindo de s podemos ir para n1 e n2, os valores de g(n1) e g(n2) são respectivamente 3 e 7. Supondo que A* expanda n1, sucedido por n2 e n3, nesse ponto g(n3)=3+2=5,o valor de g(n2) é diminuído pois um caminho de menor custo foi encontrado 3+3=6, o valor de g(n1) continua sendo 3.

O próximo passo seria estimar o h(n), podem essa função vai depender muito do domínio do problema, muitos problemas de encontrar o menor caminho entre dois pontos de um grafo possuem alguma "informação física"que pode ser usada de alguma forma para estimar o h(n), podemos em nosso exemplo, se definirmos como cidades ligadas por estradas, h(n) poderia ser a distancia aérea da cidade n até a cidade objetivo, essa distancia seria menor do que qualquer estrada da cidade n até o objetivo (HART; RAPHAEL, 1968).

Em jogos digitais e buscas em tempo real é muito comum trabalhar em cima de mapas em forma de matrizes bidimensionais, esse será o foco dos algoritmos que trataremos, nesses casos fica simples tratarmos o h(n) com a distância Euclideana ou Manhattan entre n e o destino. (BJöRNSSON et al., 2005a)

Quando definimos o mapa de busca em uma matriz bidimensional NxM, situação comum em jogos digitais e problemas de busca em tempo real (GRAHAM, 2003) (SANTOS, 2012a) (MACHADO et al., 2011), precisamos definir os custos de movimentação entre os vizinhos do ponto n, e a função heurística h(n). O custo de movimentação na grade pode variar basicamente em dois tipos, levando as diagonais em consideração ou não.

O modelo em *tile*, onde o movimento do agente está restrito a quatro direções ortogonais com custo 1, e *octile*, onde o agente pode ainda mover-se na diagonal com custo equivalente ao valor da hipotenusa de um triângulo retângulo com catetos igual a 1, sendo assim $\sqrt{2} \approx 1.4$. É comum para fim de simplificar os cálculos multiplicar os valores de custo por 10 ou 100 como pode ser visto na Figura 3.

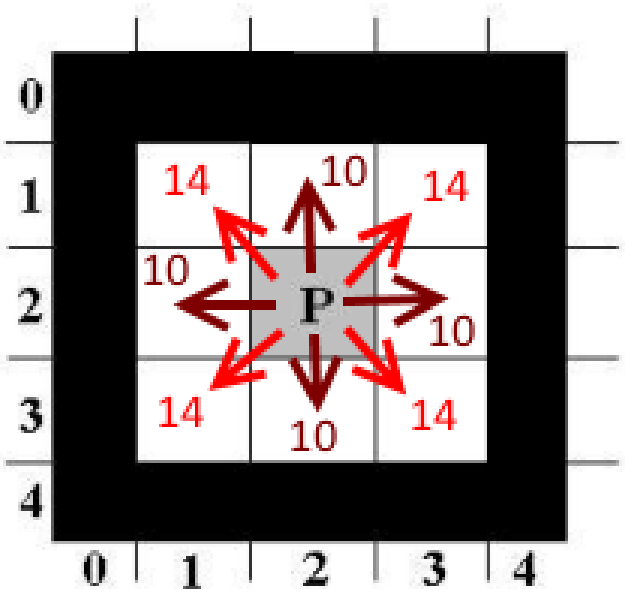

Figura 3 – Custo de movimentação

Uma heurística famosa e muito utilizada é conhecida como distância Manhattan, usada principalmente em ambientes de grades, ela estima a distância real entre os nós pela soma do deslocamento vertical e horizontal entre dois pontos, ignorando qualquer obstáculo. A distância Manhattan garante a solução com mais baixo custo, pois cada nó deve caminhar pelo menos, a sua distância Manhattan, e o movimento é dado de um nó por vez (KORF, 2000).

Ainda pode ser facilmente modificado para levar em conta o produto vetorial entre o vetor do ponto inicial e o vetor de objetivo, resultando em um caminho reto entre dois nós. (PATEL, 2010).

Ainda pode ser realizada uma alteração no algoritmo de distância Manhattan para levar em conta as diagonais, este fica chamado de distância diagonal, adicionando um custo de movimentação diagonal diferente ao de movimentação adjacente.

Também pode ser utilizada como função heurística a distância euclideana, no entanto, neste caso teremos problemas com o uso de A*, pois a função de custo g não irá corresponder a função heurística h, ja que a distância euclidiana é mais curta do que Manhattan ou a distância diagonal, você ainda terá os caminhos mais curtos, mas o A* levará mais tempo para ser executado (PATEL, 2010).

Podemos verificar na Figura 4, exemplos de distância Manhattan comparado com a distância euclideana.

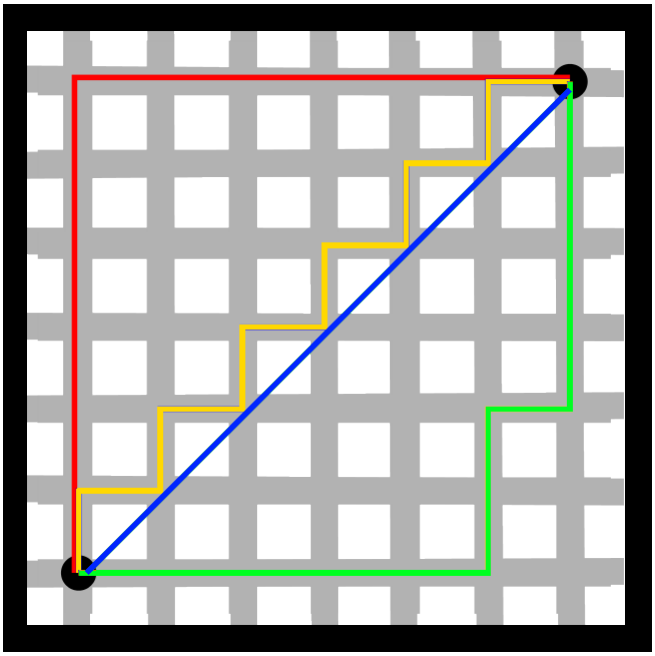

Figura 4 – Ilustração de diferença entre distância Manhatam e euclideana

Figura 4 As linhas de cor vermelho, verde e amarelo tem o mesmo tamanho (12), e representam a distancia Manhattan, a amarela leva em consideração o produto vetorial. A linha azul representa a distancia euclideana $\sqrt{72} \approx 8.485$

Uma limitação do A* é o fato requerer uma grande quantidade de recursos da CPU. Caso haja muitos nós a pesquisar, como é o caso em grandes mapas que são populares em jogos recentes, isso pode causar pequenos atrasos. Esse atraso pode ser agravado caso haja múltiplos agentes de inteligência artificial ou quando o agente tem que se mover de um lado do mapa para o outro

Este alto custo de recursos da CPU pode fazer com que o jogo congele até que o caminho ideal seja encontrado. Game designers tentam superar esses problemas realizando ajustes nos jogos, de forma a tentar evitar estas situações (CAIN, 2002).

Pelo motivo do alto custo, a heurística fornece uma expansão considerável dos nós, que são mantidos na memória durante todo o processamento. A Figura 5 mostra a expansão da heurística.

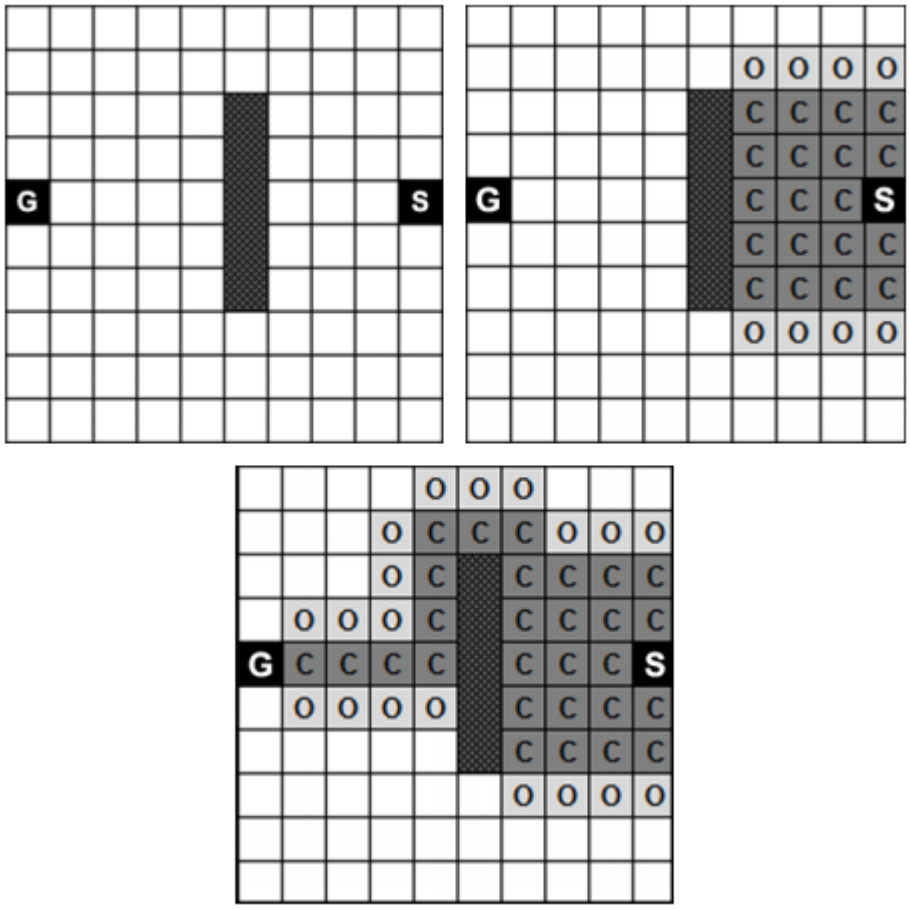

Figura 5 – Exemplo de A (SANTOS, 2012a)*

É comum no meio dos jogos digitais termos mais de um agente rodando ao mesmo tempo, um otimização para esse problema, proposta por Ko- Hsin, Cindy Wang e Adi Botea, examina o planejamento de caminhos multi-agente. Eles propõem um algoritmo chamado de FAR (*Flow Annotation Replanning*) para combater os problemas técnicos envolvidos em pesquisas globais por várias entidades. FAR implementa restrições de fluxo para evitar colisões frontais, para assim reduzir o espaço de busca que uma entidade deve considerar quando planejar a sua rota. Esse método tem a possibilidade de entrar em *deadlock*, um estado de bloqueio, porem o algoritmo é capaz de contornar esse problema (WANG; BOTEA, 2008).

É comum a utilização do BFS apenas, como possui uma função heurisitica mais simples consome menos memoria, porem não garante a certeza do melhor caminho. (SANTOS, 2012b)(RUSSELL; NORVIG, 2003).

Outra solução é o uso de metaheurísticas para a adaptação do algoritimo de busca ao ambiente. São estratégias de alto nível que guiam os processos de pesquisa, e tem

como objetivo explorar, de forma eficiente, o espaço de busca, a fim de encontrar soluções ideais tanto para problemas simples, quanto complexos. São técnicas não determinísticas e podem utilizar heurísticas para guiar seu processo de pesquisa. São exemplos de metaheurísticas, Algoritmos Genéticos (AG), BuscaTabu, Colônia de Formigas, GRASP, Simulated Annealing, dentre outras. (BLUM; ROLI, 2003)

Korf propôs o algoritmo Iterative Deepening A* (IDA*) (KORF, 1985), o mesmo utiliza a busca em profundidade combinada com a heurística do A*. O resultado é um algoritmo que encontra a solução ótima, com o gasto mínimo de memória. Essa tecnica tem como desvantagem gerar uma árvore na busca a cada iteração. Com isso, o algoritmo exige muito processamento, o que pode tornálo mais lento que o A* em determinados casos.

Em contraponto temos a proposta apresentada por Björnsson (BJöRNSSON et al., 2005b), o Fringe Search, visa otimizar o algoritmo IDA*. O foco é eliminar a principal deficiência do IDA* já citada. De forma que a cada iteração, não é mais necessário gerar a árvore novamente, pois o algoritmo armazena os nós folhas da mesma, o que faz que, na próxima iteração, a busca se inicie a partir desses nós. Com isso, o algoritmo Fringe Search consegue ser mais rápido que o IDA*.

O Recursive Best–First Search (RBFS), proposto por Korf (KORF, 1992), é um algoritmo que trabalha com a redução do consumo de memória para o processo de busca de caminhos. Este algoritmo utiliza a mesma heurística do BFS e promove uma otimização espacial para o mesmo.

A técnica de reconhecimento de padrão de obstáculos do ambiente é um assunto tratado por Demyen e Buro (DEMYEN; BURO, 2006). O algoritmo por eles proposto, chamado de Triangulation Reduction A* (TRA*), utiliza uma abstração do ambiente a fim de determinar o padrão dos obstáculos do ambiente. Este visa otimizar o tempo gasto no processamento do algoritmo A*. Esse método tem como desvantagem, requisitar um formato específico dos obstáculos do ambiente, que devem ser poligonais, o que pode inviabilizar sua utilização em qualquer tipo de ambiente.

A arte de busca de caminhos está longe de um trabalho perfeito e muito ainda precisa ser feito (PONTEVIA, 2008).

2.1.3 Aplicações

Aplicações para os algoritmos de busca podem ser diversos, como por exemplo, em um simulador de carro corrida (WANG; LIN, 2012). Utilizando o algoritmo do A* com duas modificações para encontrar o melhor caminho enquanto evita os obstáculos entre o ponto de início e o ponto de destino. A primeira modificação consiste em utilizar o teorema de Pitágoras, onde primeira calcula a distância entre dois pontos(1 e 2), verifica

se existe algum obstáculo dentre eles, se existir, utiliza o terceiro ponto para calcular a hipotenusa e verifica se existe obstáculo entre a hipotenusa, se não existir, remove o ponto 2 e começa a considerar o caminho do ponto 1 para o ponto 3. A segunda modificação consiste em somente ir para frente, isso significa, procura somente os pontos a frente do carro, direita, esquerda e frente, simulando um controle de carro.

O projeto utiliza 3 pistas reais de corrida da Formula 1, Peru, Itália e Hungria, sendo cada pista, uma imagem de escala 1280x782 *pixels* retiradas do site oficial da Formula 1. O Carro é implementado em *Microsoft XNA Game Studio*, plataforma usada para desenvolver jogos para *Windows Phone, Xbox e Windows*. As Imagens das pistas são modificadas para serem mapas de detecção de colisão, a pista é pintada de preto, indicando onde o carro pode andar e o resto pintado de branco indicando os o carro não pode andar. O carro tem o tamanho de 18x12 *pixels* e uma movimentação de 2 *pixels* por segundo. Como o XNA trabalha com uma taxa de quadros de 60 quadros por segundo, o carro se movimenta a 120 *pixels* por segundo.A Figura 6 mostra um exemplo de rota gerada na pista do Peru.

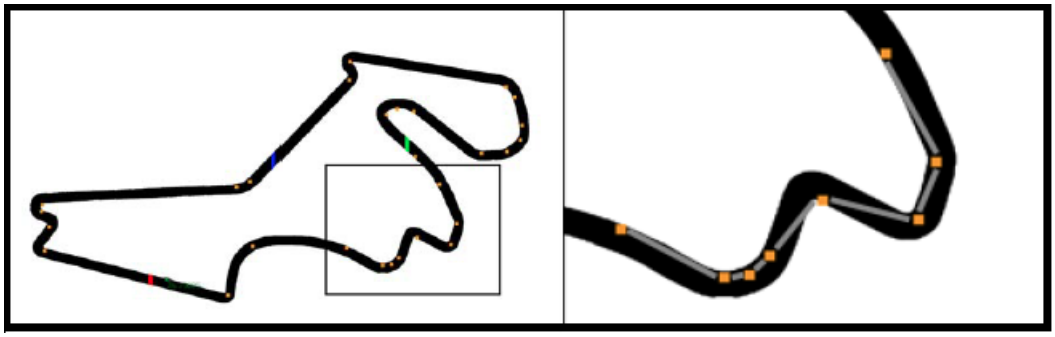

Figura 6 – Pisca de formula 1 do Peru (WANG; LIN, 2012)

Os resultados obtidos por (WANG; LIN, 2012) na primeira modificação conseguiu economizar ciclos de CPU, reduzindo o numero de pontos indicadores da pista em 97%. A segunda modificação tem a vantagem de obter o tempo de volta mais curto, por que reduz o numero de nós do algoritmo A* de 4 para 3 nós]. A desvantagem é que o carro pode balançar em curvas acentuadas. Em geral, as modificações garantiram uma melhoria em performance, economizando o numero de ciclos de CPU.

2.2 Algoritmo paralelo de busca

O problema de busca de caminho pode ser facilmente quebrado em duas maneiras (MILLER, 2010).

A primeira abordagem seria iniciar a busca de ambos os nós, inicial e final simultaneamente, Esta variação do A*, também conhecida como Pesquisa Bidirecional, permite que um problema seja dividido em dois núcleos, mas não escala facilmente com um maior numero de núcleos ou com um problema de maior tamanho. Esta abordagem requer dois

núcleos livres por busca, e não seria capaz de executar em um ambiente que não ofereça no mínimo essa estrutura. (CHANG, 2009).

A segunda abordagem seria tratar cada busca de caminho como uma unidade individual e distribuir cada uma para um núcleo. Esta abordagem funciona facilmente com qualquer número de núcleos e é possivelmente a forma mais eficaz de distribuição de grandes conjuntos de dados (CHAMPANDARD, 2010) (MILLER, 2010).

2.3 Algoritmos genéticos

AG é uma técnica amplamente utilizada de IA, que utilizam conceitos provenientes do princípio de seleção natural para abordar uma ampla série de problemas, geralmente de adaptação. (LUCAS, 2002)

2.3.1 Funcionamento

Inspirado na maneira como o seleção natural explica o processo de evolução das espécies, Holland (HOLLAND, 1975) decompôs o funcionamento dos AG em sete etapas, essa são *inicialização*, *avaliação*, *seleção*, *cruzamento*, *mutação*, *atualização* e *finalização* conforme a Figura 7.

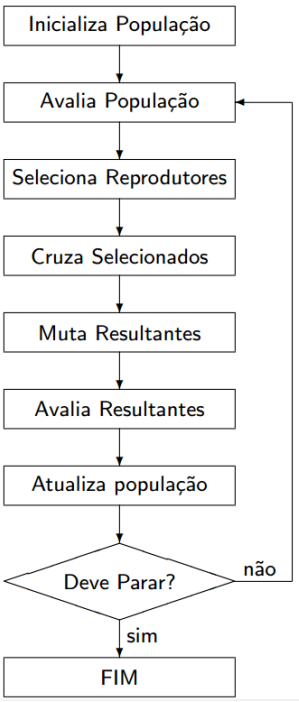

Figura 7 – Estrutura de um AG (LUCAS, 2002)

2.3.2 Inicialização

Criar uma população de possíveis respostas para um problema. É comum fazer uso de funções aleatórias para gerar os indivíduos, sendo este um recurso simples que visa

fornecer maior diversidade.

2.3.3 Avaliação

Avalia-se a aptidão das soluções, os indivíduos da população, então é feita uma análise para que se estabeleça quão bem elas respondem ao problema proposto. A função de avaliação também pode ser chamada de função objetivo. Ela pode variar de acordo com problema, Calcular com exatidão completa o grau de adaptação dos indivíduos pode ser uma tarefa complexa em muitos casos, e se levarmos em conta que esta operação é repetida varias vezes ao longo do processo de evolução, seu custo pode ser consideravelmente alto. Em tais situações é comum o uso de funções não determinísticas, que não avaliam a totalidade das características do indivíduo, operando apenas sobre uma amostragem destas.

2.3.4 Seleção

Ela é a responsável pela perpetuação de boas características na espécie. Neste estágio que os indivíduos são escolhidos para posterior cruzamento, fazendo uso do grau de adaptação de cada um é realizado um sorteio, onde os indivíduos com maior grau de adaptação tem maior probabilidade de se reproduzirem. O grau adaptação é calculado a partir da função de avaliação para cada individuo, determina o quão apto ele esta para reprodução relativo a sua população.

Selection Random: Gera um numero aleatório entre 0 e o tamanho total da população e retorna o indivíduo do índice escolhido.

Selection Roulette Wheel: Faz a soma de todos os valores da função de aptidão da população, depois calcula a porcentagem de cada indivíduo referente ao total e guarda em um vetor. Então é gerado um valor A aleatório entre 0 e 1 e multiplicado pelo valor total dos pesos. Para selecionar o indivíduo é feito um loop nos pesos e seus valores somados até que o valor A seja igual ou menor que zero, o índice do peso que fez a condição acontecer, se o índice do indivíduo selecionado. Desta forma aumentando a possibilidade de selecionar um indivíduo com maior aptidão.

2.3.5 Cruzamento

Características das soluções escolhidas na seleção são recombinadas, gerando novos indivíduos.

CrossOver Simple: Utiliza dois indivíduos selecionados, define dois números aleatórios de 0 até menor tamanho da lista de cromossomos entre os dois, sendo que o primeiro índice tem que ser menor que o segundo índice e os mesmos não podem ser iguais. Esse tamanho é utilizado para trocar cromossomos entre os dois indivíduos, ou seja,

adicionar todos os cromossomos do primeiro indivíduo do índice igual ao primeiro numero, até o índice segundo numeSAro, e repete o processo contrario.

Crossover OBX (Order-Based Crossover): Utiliza dois indivíduos escolhidos na seleção, então define dois números aleatórios, de 0 até menor tamanho da lista de cromossomos entre os dois, sendo que o primeiro tem que ser menor que o segundo e não podem ser iguais. O primeiro numero até o segundo numero, são definidas posições aleatórias e são salvas em uma lista. Faz um loop na lista e troca o cromossomo da posição do primeiro indivíduo para o segundo e do segundo para o primeiro.

Crossover PBX (Position-Based Crossover): Utiliza dois indivíduos selecionados, então define dois números aleatórios, de 0 até menor tamanho da lista de cromossomos entre os dois, sendo que o primeiro índice tem que ser menor que o segundo índice e os mesmos não podem ser iguais. Entre esse tamanho são definidas posições aleatórias e guardadas em uma lista. Os indivíduos resultantes são zerados, e para cada posição é trocado do cromossomo principal para o resultante de mesma posição outro da mesma posição. As posições não preenchidas são completadas com os cromossomos restante, seguindo a ordem do cromossomo e adicionado se ele não ja existir na lista.

2.3.6 Mutação

Características dos indivíduos resultantes do processo de reprodução são alteradas, acrescentando assim variedade a população. A mutação opera sobre os indivíduos resultantes do processo de cruzamento e com uma probabilidade pré-determinada efetua algum tipo de alteração em sua estrutura. A importância desta operação é o fato de que uma vez bem escolhido seu modo de atuar, é garantido que diversas alternativas serão exploradas.

MutateEM (Exchange Mutation): Define duas das posições aleatórias distintas do segundo cromossomo até o ultimo, e troca os cromossomos do indivíduo.

MutateSM (Scramble Mutation): Define duas das posições aleatórias distintas do segundo cromossomo até o ultimo, e uma quantidade aleatória. Então faz um loop da quantidade aleatória e mistura os cromossomos que estão entre a posição inicial e final trocando aleatoriamente dois pontos entre eles.

MutateDM (Displacement Mutation): Define duas das posições aleatórias distintas do segundo cromossomo até o ultimo, e remove todos os cromossomo entre essa posições e recoloca a partir de uma posição aleatória.

MutateIM (Insertion Mutation): Define uma posição aleatória, remove o cromossomo da posição, reorganiza os cromossomos e insere o cromossomo removido em uma nova posição aleatória.

MutateIVM (Inversion Mutation): Define duas das posições aleatórias distin-

tas do segundo cromossomo até o ultimo, e inverte todos os cromossomos que está entre as posições.

MutateDIVM (Displaced Inversion Mutation):Define duas das posições aleatórias distintas do segundo cromossomo até o ultimo, e remove todos os cromossomo entre essa posições e recoloca a partir de uma posição aleatória de forma invertida.

2.3.7 Atualização

Os indivíduos criados no processo de reprodução e mutação são inseridos na população.

Na forma mais tradicional deste a população mantém um tamanho fixo e os indivíduos são criados em mesmo número que seus antecessores e os substituem por completo.

Existem, porém, algumas alternativas, o número de indivíduos gerados pode ser menor ou o tamanho da população pode sofrer variações e o critério de inserção pode variar, por exemplo, nos casos em que os filhos substituem os pais, ou em que estes só são inseridos se possuírem maior aptidão que o cromossomo que sera substituído, ou o manter sempre o conjunto dos n melhores indivíduos.

2.3.8 Finalização

É testado se as condições de encerramento da evolução foram atingidas, retornando para a etapa de avaliação em caso negativo e encerrando a execução em caso positivo.

Os critérios para a parada podem ser vários, desde o número de gerações criadas até o grau de convergência da população atual.

Toda base dos AG se fundamenta nos indivíduos, eles são a unidade básica em qual o algoritmo se baseia, sua função é codificar as possíveis soluções do problema a ser tratado e partir de sua manipulação no processo evolutivo, a partir daí que são encontradas as respostas.

Esses indivíduos precisam de uma representação, essa será o principal responsável pelo desempenho do programa. É comum chamar de *genoma* ou *cromossomo* para se referir ao individuo. Por essa definição podemos resumir um indivíduo pelos genes que possui, ou seja seu *genótipo*.

Apesar de toda representação por parte do algoritmo ser baseada única e exclusivamente em seu genótipo, toda avaliação é baseada em seu fenótipo, o conjunto de características observáveis no objeto resultante do processo de decodificação dos genes do individuo, ver Tabela 1.

Tabela 1 – Exemplos de genótipos e fenótipos correspondentes em alguns tipos de problemas (LUCAS, 2002)

Problema	**Genótipo**	**Fenótipo**
Otimização numérica	0010101001110101	10869
Caixeiro viajante	CGDEHABF	Comece pela cidade C, depois passe pelas cidades G, D, E, H, A, B e termine em F
Regras de aprendizado para agentes	$C_1R_4C_2R_6C_4R_1$	Se condição 1 (C_1) execute regra 4 (R_4), se (C_2) execute (R_6), se (C_4) execute (R_1)

Para cada indivíduo é calculado o seu grau de adaptação, a partir de uma função objetivo, comumente denotada como na formula 2.3.

$$f_O(x) \tag{2.3}$$

Que vai representar o quão bem a resposta apresentada pelo individuo soluciona o problema proposto.

Também é calculado o grau de adaptação do indivíduo relativo aos outros membros da população a qual ele pertence, esse é chamado de grau de aptidão, para um indivíduo x temos seu grau de aptidão denotado pela fórmula 2.4.

$$f_A(x) = \frac{f_O(x)}{\sum_{i=1}^{n} f_O(i)} \tag{2.4}$$

Sendo n o tamanho da população.

A dinâmica populacional é a responsável pela evolução, ao propagar características desejáveis a gerações subsequentes no processo de cruzamento, enquanto novas são testadas no processo de mutação.

Algumas definições importantes relativo as populações de um AG são:

Geração: É o número de vezes em que a população passou pelo processo de seleção, reprodução, mutação e atualização.

Média de adaptação: É a taxa média que ao indivíduos se adaptaram ao problema, é definida pela formula 2.5.

$$M_A = \frac{\sum_{i=1}^{n} f_O(i)}{n} \tag{2.5}$$

Grau de convergência: define o qual próxima esta a media de adaptação desta população relativo as anteriores. O objetivo dos AG é fazer a população convergir para uma valor de adaptação ótimo. Um estado negativo que pode ocorrer relativo a esta medida é a *convergência prematura*, a mesma ocorre quando a população converge em uma média de adaptação sub-ótima, e dela não consegue sair por causa de sua baixa diversidade.

Diversidade: Mede o grau de variação entre os genótipos da população. Ela é fundamental para o tamanho da busca. Sua queda esta fortemente ligada ao fenômeno de *Convergência prematura.*

Elite: São os indivíduos mais bem adaptados da população. Uma técnica comum nos AG é p *elitismo*, onde são selecionados k melhores indivíduos que serão mantidos a cada geração.

2.3.9 Aplicações

Existem vários aplicações para os algoritmo genéticos, por serem uma inteligência artificial não supervisionada, de rápido aprendizado e podendo ser paralelizado.

O modelo m-PRC(Problema de Rotas de Cobertura multi-veículo) é uma aplicação de algoritmos genéticos para construção de rotas em uma região mapeada, para encontrar uma boa distribuição de viaturas para patrulhamento urbano usado por departamentos de segurando como a policia, guardas municipais ou segurança privada (OLIVEIRA, 2009). O Modelo é definido como um grafo não direcionado 2.6.

$$G = (V \cup W, E) \tag{2.6}$$

Onde 2.7:

$$V \cup W \tag{2.7}$$

Compõem o conjunto de vértices e E o conjunto de arestas, ou seja, o subgrafo induzido por E e um grafo completo cujo conjunto de nós é V. V são todos os vértices que podem ser visitados e é composto pelo subconjunto T, que são os vértices que devem ser visitados por algum veiculo. W é um conjunto de vértices onde todos os M veículos devem passar. M é o numero de rotas de veículos que começam no vértice base V_0.

O m-PRC atribui o conjunto de m rotas de veículos com as restrições: todas as m rotas de veículos começam e terminam na base V_0, Tem exatamente m rotas, cada vértice de V pertence a no máximo uma rota, cada vértice de T pertence a exatamente uma rota,

com exceção a base, cada vértice de W deve ter uma rota que passa por ele e em uma distancia C de um vértice V visitado, O modulo da diferença entre o número de vértices de diferentes rotas não pode exceder um determinado valor R. A Figura 8 mostra o grafo da relação de V com W.

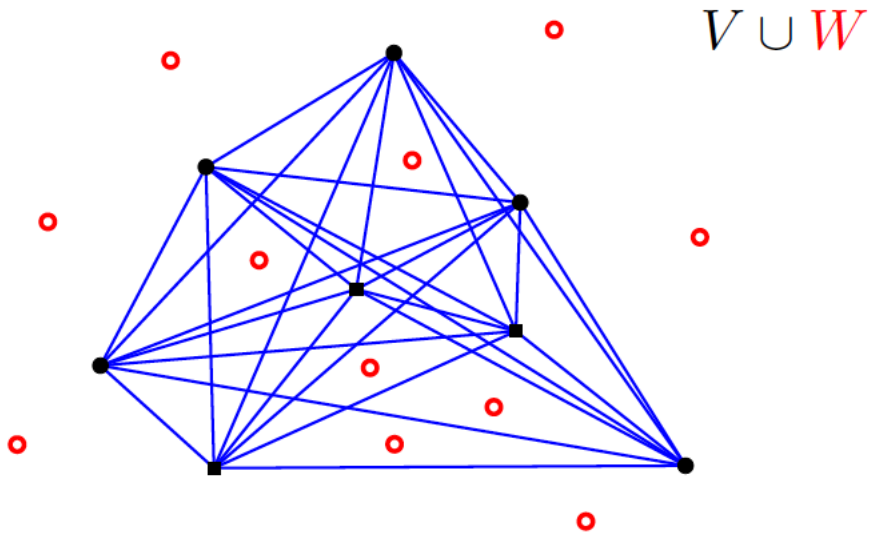

Figura 8 – Exemplo de gráfo não direcionado para V U W. (OLIVEIRA, 2009)

Para utilizar o algoritmos genéticos com o modelo m-PRC, o trabalho propõem dois modelos. O AGS (Algoritmo genético sequencial), que utiliza heurísticas GENIUS e 2-opt balanceada para ajustes finais para tentar melhor a solução; O AGH(Algoritmos genéticos H-1-PRC), que utiliza heurísticas H-1-PRC-MOD e 2-opt balanceada em todo o processo de resolução.

A conclusão de (OLIVEIRA, 2009) é que a utilização de algoritmos genéticos para a resolução de de uma adaptação do problema de rotas de cobertura de veículos como bastante relevantes e de fácil manipulação. O modelo AGS resolve o problema de forma rápida e tem uma fácil implementação dentro dos critérios de comparação adotadas. O modelo AGH é mais lento e não conseguiu encontrar a solução para alguns exemplos.

2.4 Algoritmos genéticos para busca de caminhos

Buscando melhorar a eficiência dos algoritmos de busca, foram criadas formas híbridas, onde utilizam inteligência artificial para analisar todo o percusso e ajudar o algoritmo de busca a decidir em momentos que a próxima ação é incerta.

Utilizando o algoritmo de busca BFS (*Best First Search*), foi desenvolvido o modelo PPGA (*Patterned based Pathfinding with Genetic Algorithm*). O modelo utiliza algoritmos genéticos como um módulo para calcular os sub-caminhos ao longo do processo de busca, cada vez que o módulo é chamado, herda informações da chamada anterior, tendo um ganha considerado de desempenho. O módulo é chamado, quando nenhum dos valores

dados pelo BFS são melhores que o valor atual. A Figura 9 mostra uma fluxograma do funcionamento do modelo.

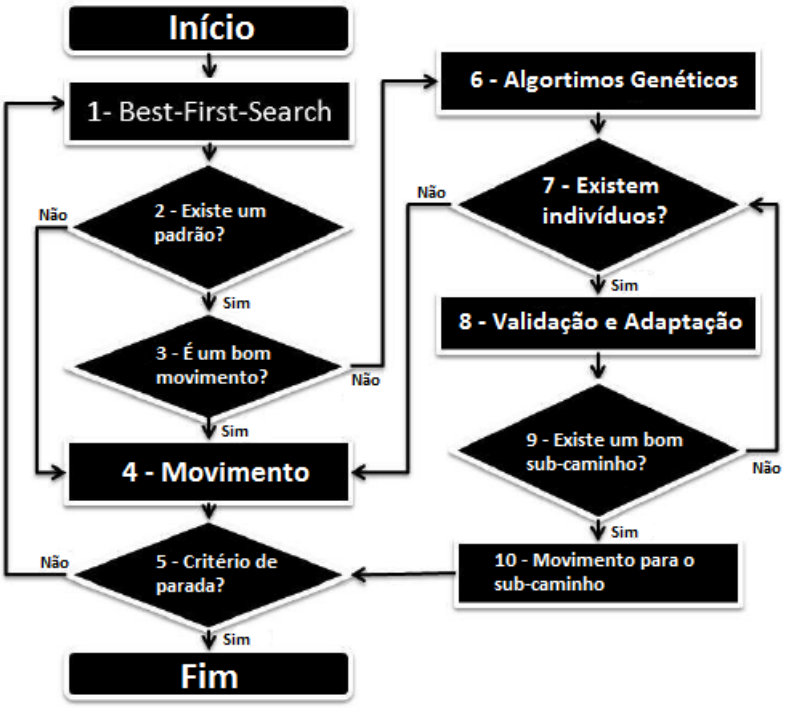

Figura 9 – Fluxograma do PPGA (SANTOS, 2012a)

O modelo PPGA é dividido em 10 etapas:

1. **Best-First-Seach** - Seleciona o nó atual e seleciona o menor valor na lista de nós abertos.

2. **Existe um padrão?** - No modulo de AG existe uma contagem de proporção de vezes que sub-caminho são utilizados pelas tentativas de utilização, se essa proporção for baixa, é definido que não existe um padrão na busca, então, não utiliza o modulo de AG.

3. **É um bom movimento?** - Verifica se o nó encontrado tem o valor menor que o atual, isto é, se aproxima mais do destino.

4. **Movimento** - Adiciona o nó atual para a lista fechada.

5. **Critério de parada?** - Se o ó atual é o nó de destino, o algoritmo termina, se não, continua para a próxima iteração.

6. **Algoritmo genético** - É obtida apenas uma geração com os seguintes critérios:

   a) A população inicial tem 4 indivíduos.

   b) Gera os primeiros descendentes utilizando com 4 primeiros indivíduos com um critério de 50%.

c) Gera os próximos descendentes utilizando, pela mutação da população inicial com um critério de 25%.

d) É calculado a aptidão para 12 indivíduos. O elemento utilizado na ultima geração recebe o valor valor.

7. **Existem indivíduos?** - Verifica se existem indivíduos para serem avaliados na população. Se nenhum dos indivíduos forem validos, o algoritmo continua a manutenção do nó atual.

8. **Validação e adaptação** - Ajuda a garantir ageração e adaptação de bons indivíduos.

9. **Existe um bom sub-caminho?** - Os indivíduos gerados são testados, um em cada iteração, se não for um bom elemento, 'removido da lista da população.

10. **Movimento para o sub-caminho** - O elemento será usado como um sub-caminho composto pelos nós que serão adicionados à lista fechada. O nó real se tornar o último nó do sub-caminho. Este elemento será inserido para a próxima geração do GA.

A Figura 10 mostra o resultado de caminho percorrido pelo BFS (A) e o PPGA (B) em um mesmo ambiente simulado.

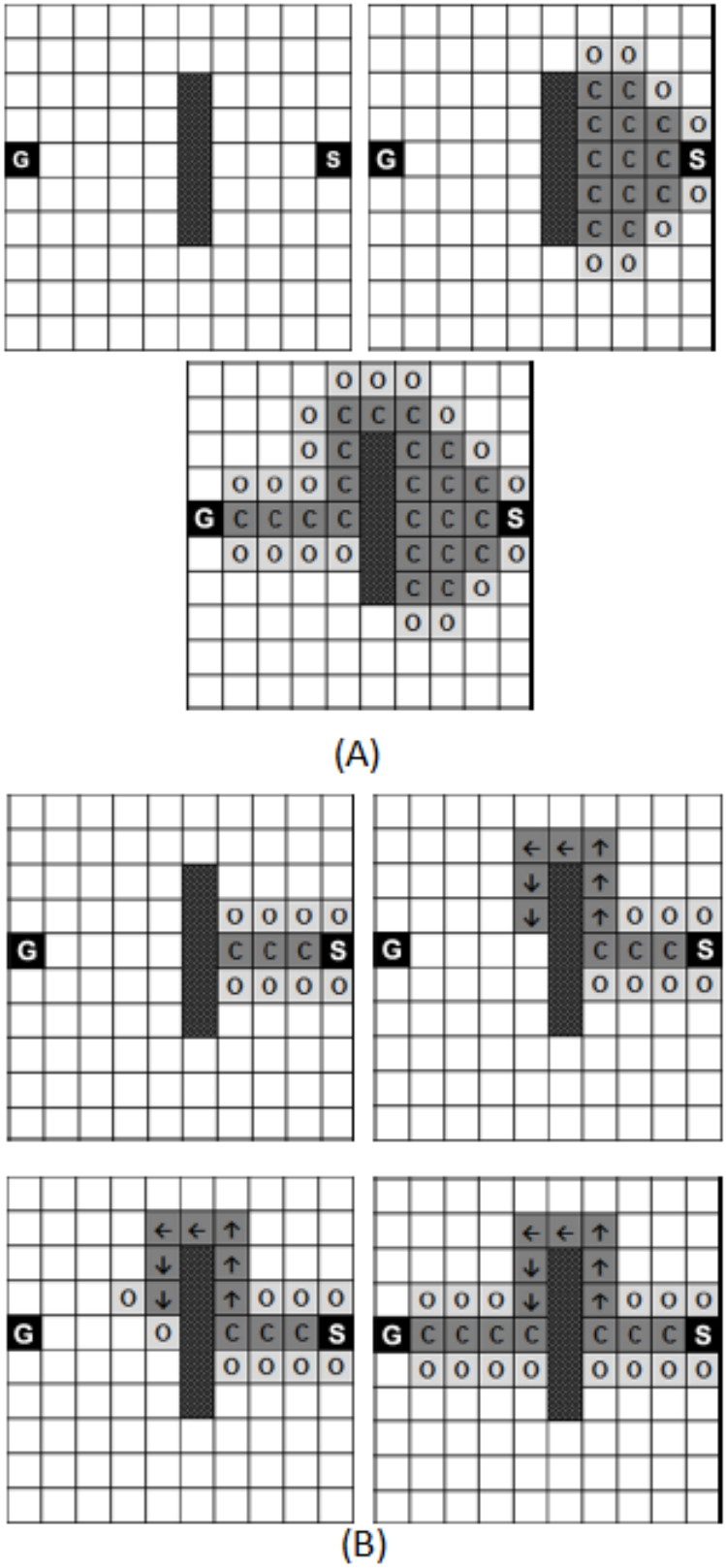

Figura 10 – Caminho criado pelo BFS e o PPGA (SANTOS, 2012a)

Os resultados obtidos da análise feita por (SANTOS, 2012a) do PPGA foram bons, por encontrar boas soluções em um curto período de tempo, mostrando ser muito bom para mapas onde o percurso segue um padrão, mas perdendo para o BFS para mapas

mistos ou que não seguem o mesmo padrão ao longo o percurso, considerando o modelo como promissor, indicando que mudanças nos parâmetros do algoritmo genéticos, pode melhorar o desempenho dos testes (SANTOS, 2012a).

2.5 Algoritmos genéticos paralelos ou distribuídos para busca de caminhos

Foi demonstrado que algoritmos genéticos paralelos são eficientes para a resolução de problemas de busca de caminho, tal como o clássico problema do caixeiro viajante, que consiste em dado um número finito de cidades com seus custos de viagem entre elas, deve-se encontrar o caminho mais curto para viajar entre todas as cidades e voltar ao ponto inicial.

O problema pode ser representado pelo modelo de um grafo direcionado ponderado, aplicando a mesma ideia, o problema seria encontrar o caminho de menor custo para percorrer todos os nós, de maneira análoga, as cidades seriam os nós e a distancia entre elas, o peso das arestas (D.LOHN SILVANO P. COLOMBANO, 2000)(ALAOUI; EL-GHAZAWI, 2000)(MUHLENBEIN, 2000).

Os algoritmos genéticos exigem apenas o valor dado por uma função objetivo como parâmetro e mesmo sobre espaços de busca grandes, tem uma convergência rápida. Por causa do processo associado, agrega uma visão mais global do espaço de busca na prática de otimização e possuem uma fácil paralelização por causa da independência dos seus processos. Em comparação com as técnicas de busca mais comuns,que requerem informações derivadas, continuidade do espaço de busca ou conhecimento completo da função objetiva.(MOLE, 2002).

A solução para este tipo de problema pode requer uma quantidade grande de processamento. Uma boa solução seria dividir o processamento do problema em pequenas partes e distribuir cada parte para um processador a parte, trabalhando de forma distribuída ou paralela. Vários modelos para essa finalidade foram propostos.

Um modelo interessante para paralelização seria o de mestre-escravo, onde o mestre fica responsável na manutenção da população e execução dos operadores genéticos. A avaliação dos melhores indivíduos é distribuída para os demais escravos. O mestre envia um indivíduo a cada um dos escravos subjacentes.

Cada escravo realiza a interpretação do problema, aplica a função de cálculo para a escolha dos melhores indivíduos e envia seus resultados ao mestre, que executa seleção dos indivíduos e a geração da nova população, repetindo o processo como um todo. Essa estrutura teve implicação satisfatória para a automação de design de circuitos eletrônicos(D.LOHN SILVANO P. COLOMBANO, 2000). A Figura 11 mostra o desenho

do modelo proposto.

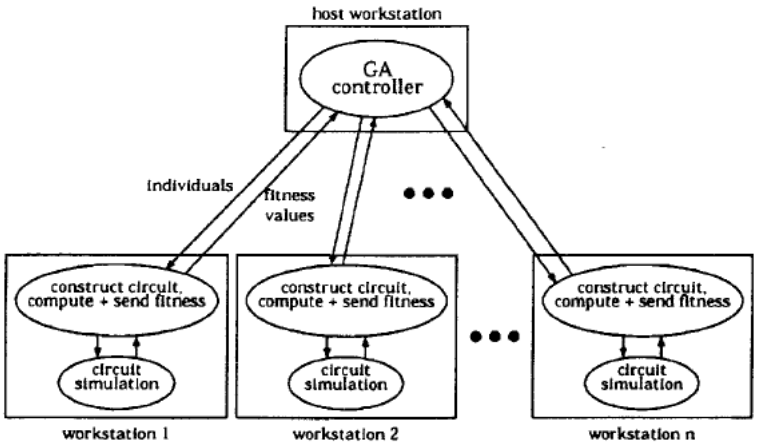

Figura 11 – Desenho do modelo de (D.LOHN SILVANO P. COLOMBANO, 2000)

Outra forma de trabalhar com o modelo de mestre escravo, seria definir que cada um dos nós escravos subjacentes fica responsável por sua própria população. O nó central mestre, cria as populações iniciais e as distribui para os nós escravos. Cada nó escravo processa a evolução da população por um determinado número de gerações e então a submete ao mestre. O mestre então seleciona os melhores indivíduos dentre todas as populações dos nós escravos e os distribui novamente.

Em cada nó escravo, os novos indivíduos distribuídos pelo mestre são inseridos na população corrente e o processo de evolução recomeça. A migração entre os escravos, que é controlado pelo nó mestre, implementa o mecanismo que regula a velocidade da convergência e oferece os meios de escape dos mínimos locais. Entretanto a migração das populações dos nós escravos para o mestre e vice versa pode impor um certo grau de sobre carga, dependente do meio de comunicação entre os nós. Esse modelo obteve sucesso no mapeamento de tarefas em maquinas paralelas. (ALAOUI; EL-GHAZAWI, 2000)

Podemos partir do ponto que cada indivíduo é o responsável por encontrar e reproduzir com um parceiro em sua vizinhança. O controle de seleção e reprodução se espalha pela população e o algoritmo deixa de ser centralizado em um mestre, com isso, diminui o grau de sincronização e facilita a paralelização.

O processo do algoritmo é definir uma representação genética para o problema e criar a estrutura de vizinhança e sua população inicial. Cada indivíduo faz uma busca em sua vizinhança e seleciona um parceiro para a reprodução. Uma geração descendente é criada com o operador genético resultante. (MUHLENBEIN, 2000)

Podemos observar alguns problemas nos modelos apresentados (MOLE, 2002), no modelo de (D.LOHN SILVANO P. COLOMBANO, 2000) existe problema em explorar o paralelismo no cálculo de verificação dos indivíduos não explorando para a reprodução e mutação. No modelo de (MUHLENBEIN, 2000), tem a possibilidade de utilizar vários

métodos de busca de indivíduos da mesma população, sendo úteis em casos que a eficiência dos métodos de busca se mostram dependentes da instancia do problema.

O modelo (ALAOUI; EL-GHAZAWI, 2000), por todos os escravos devem enviar para o nó mestre, demanda uma grande capacidade de processamento no nó mestre, e proporciona a divisão das populações em pequenas ou de médio porte.

(MOLE, 2002) desenvolveu seu próprio modelo, utilizando o modelo de (ALAOUI; EL-GHAZAWI, 2000) como inspiração. O modelo segue o conceito mestre-escravo, o mestre crias as populações e distribui a cada uma delas, os conjuntos de genes e parâmetros iniciais. O mestre é utilizado para a troca de indivíduos entre as populações, mantendo um indivíduo de cada população até serem substituídos por um melhor e envia esses indivíduos para as populações que não seja a sua de origem.

As populações são independentes, gerando seus indivíduos iniciais com base nos genes enviados pelo mestre, aplicando seus próprios operadores de evolução e a população que determina os parceiros dos indivíduos. A Figura 12 mostra o desenho do modelo.

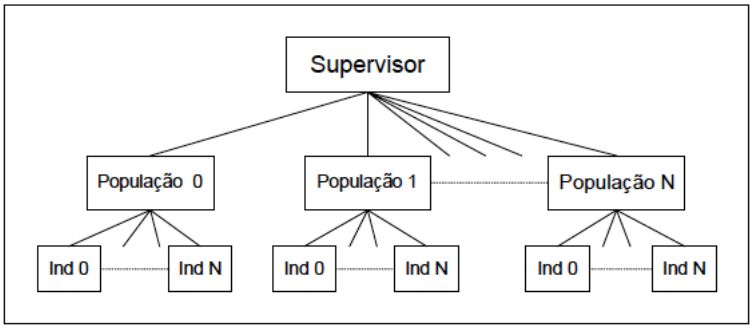

Figura 12 – Protótipo de (MOLE, 2002)

3 Proposta

A proposta deste trabalho é um modelo de busca de caminho do algoritmo BFS com IA em uma arquitetura paralela. O BFS tem sua utilização muito popular no ramo de jogos eletrônicos e existe uma necessidade na melhoria do desempenho destes algoritmos nesse meio. (GRAHAM, 2003)

A primeira parte do modelo que iremos propor, trata-se da utilização de inteligência artificial, em especifico, algoritmos genéticos. Uma arquitetura híbrida foi demonstrada indicando sucesso na melhoria de desempenho. (LEIGH; MILES, 2007)

Utilizar algoritmos genéticos de forma paralela mostra uma grande melhoria de desempenho conforme o numero de *threads* é aumentado. (ROSHANI, 2015)

Para a utilização em jogos eletrônicos é preciso que o modelo execute em tempo real, recalculando trajeto a cada iteração gráfica, tentando a partir daí, encontrar a uma solução mais próxima da ótima. Uma forma em tempo real do A* utilizando algoritmos genéticos teve resultados positivos. (MACHADO et al., 2011)

4 Metodologia

Para realizar os testes precisamos de uma grande quantidade de mapas. Para isso desenvolvemos uma ferramenta que os gera de forma automática a partir de parâmetros previamente definidos.

Nestes parâmetros definimos um tamanho $N \times M$ para o mapa, a densidade percentual que o mapa será coberto por obstáculos, se haverá ou não movimentação diagonal e se queremos um mapa gerado de forma totalmente aleatória ou que seja gerado repetindo um padrão de bloco de tamanho $L \times L$. O tamanho L do bloco que será repetido também é definido na configuração do gerador, é gerado de forma aleatória nos mesmos padrões do mapa totalmente aleatório.

Também é definido o tamanho mínimo do caminho solução do mapa, ou seja, o tamanho do caminho entre o ponto inicial e final. Cada mapa gerado é utilizado o algoritmo A* para verificar se é solúvel e o tamanho do caminho maior ou igual ao tamanho de caminho mínimo definido nos parâmetros. O caminho mínimo é importante para evitar que seja gerado mapas onde o ponto inicial e final estão muito próximos.

Pelo fato de ter pouco tempo para a coleta dos dados, não foi possível analisar todas as possibilidades das configurações do GA. Como pode ser visto na tabela a baixo.

Em mapas 100x100 podem existir momentos que o GA não está encontrando uma solução, com isso faz com que ele demore para chegar no limite de gerações, que está definido em 1000. Com uma média de 30 segundos por mapa, todas as configurações existe na implementação se torna inviável para o período de tempo para a finalização do projeto.

Tabela 2 – Possibilidades do GA

Nome	**Total**	**Escolha**
Tamanho	1	2
Tipo Mapa	2	2
Movimentação	4	2
Qt. Mapas	100	50
Qt. Repetição GA	10	4
Heurísticas	4	4
Fitness	4	3
Cruzamento	3	2
Mutação	7	2
Seleção	2	1
Total	**5.376.000**	**76.800**
Tempo Médio	30	30
Dias	**1866,67**	**26,67**

Tabela 3 – Comparação Fitness

	Média de Tempo (ms)		**Não Encontrou**	
Fitness	**10x10**	**100x100**	**10x10**	**100x100**
FitnessHeuristic	**106**	62.542	**3**	**59**
FitnessWithCirclicValidation	991	**34.119**	91	191
FitnessWithCollisionDetection	1.467	**30.405**	105	202
FitnessWithCollisionDAndCV	**510**	44.943	**21**	**103**
Total	**768**	**43.002**	**220**	**555**

Tabela 4 – Comparação Mutação

	Média de Tempo(ms)		**Não encontrou**	
Mutações	**10x10**	**100x100**	**10x10**	**100x100**
MutateBitwise	938	42.755	50	76
MutateDIVM	**753**	**42.161**	**26**	**87**
MutateDM	**617**	44.924	**20**	**80**
MutateEM	781	43.362	35	74
MutateIM	728	43.324	25	73
MutateIVM	813	44.133	34	85
MutateSM	749	**40.356**	30	80
Total	**768**	**43.002**	**220**	**555**

Tabela 5 – Comparação Cruzamento

	Média de Tempo(ms)		**Não encontrou**	
Rótulos de Linha	**10x10**	**100x100**	**10x10**	**100x100**
CrossoverOBX	**167**	56.548	5	**121**
CrossoverPBX	1.777	**33.921**	200	259
CrossoverSimple	**362**	**38.537**	15	**175**
Total Geral	**768**	**43.002**	**220**	**555**

Para a decisão de como ficaram os parâmetros finais, utilizamos dois tipos mapas, pequeno 10x10 e grande 100x100, sendo gerados 10 mapas de cada 1 vez para as configurações do GA, como o tempo é o melhor problema, ele foi o parâmetro para a tomada de decisão.

Decidimos escolher 2 mais rápidas das 7 mutações , 2 mais rápidos dos 3 Cruzamentos e como a Fitness é o processo muito importante do GA, as 3 mais rápidas das 4 Fitness. Reduzimos de 10 para 4 o número de vezes que o GA vai rodar sobre o mesmo mapa e o numero todos de mapas para de 100 para 50. Como tivemos diferentes resultados com as Fitness para tamanho diferentes de mapas, decidimos por 2 tipos de tamanho de mapa, 20x20 e 100x100.

Para a análise foram gerados mapas de tamanho 100×100 com um caminho mínimo mínimo de 15 passos e uma densidade de obstáculos de 30%. No total foram gerados 400 mapas, sendo divididos em 200 gerados a partir de um padrão e 200 gerados aleatoriamente. Para cada mapa com padrão é gerado aleatoriamente um bloco de 5×5 que é repetido

até completar o tamanho total do mapa.

Os 200 mapas de cada tipo são divididos em 2, de forma que sejam 100 mapas para que permitem movimentação diagonal (figura:14) e 100 que não permitem (figura:13).

A configuração de movimentação diagonal interfere diretamente na heurística utilizada e na geração do mapa, o mesmo pode ou não ter uma solução dependendo do tipo de diagonal. Logo o mapa deve ser gerado levando isso em consideração.

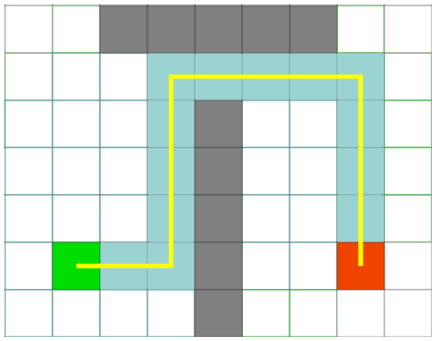

Figura 13 – Não permite andar nas verticais, podendo somente se movimentar para cima, baixo, direita e esquerda.

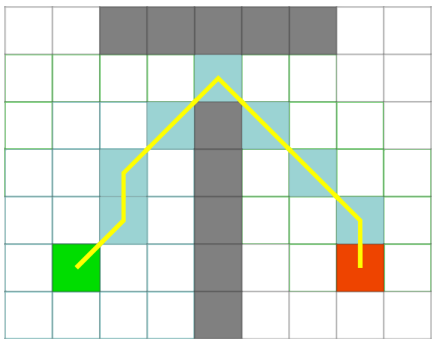

Figura 14 – É permitidos seguir na vertical, se um dos lados estiver livre.

Os algoritmos clássicos de busca que utilizamos para comparação são A*, BFS, Dijkstra e IDA*, sendo executados uma vez para cada uma das heurísticas selecionadas para os testes, essas são Manhattam, Euclideana, Octil e Chebyshev.

Devido à natureza não determinística do Algoritmo genético rodamos ele 5 vezes para cada mapa, e calculamos a média de tempo, custo de memória e tamanho de caminho resultado para comparação com os algoritmos clássicos.

As possíveis configurações do GA dependem dos operadores de cruzamento, mutação e seleção, alem da função de aptidão e a heurísticas utilizada por ela. Os operados de cruzamento implementados para a analise são *"Simples"*, OBX e o PBX. Os operadores de Mutação são Bitwise, DIVM, DM, EM, IM,IVM e SM. Na seleção utilizamos o algoritmo de Roleta. A função de aptidão é baseada nas heurísticas implementadas, sendo uma a heurística sem nenhuma alteração, outra implementando a heurística e penalizando caminhos cíclicos, outra implementando a heurística e penalizando encontro com paredes

e a ultima implementando a heurística e penalizando tanto caminhos cíclicos quanto encontro com paredes.

Para os testes de GA, utilizamos a heurística Manhattam para movimentação sem diagonal, e Octile para movimentação com diagonal, a escolha é devido ao fato da heurística ser fortemente ligada a forma de movimentação como é citado no artigo (PATEL, 2010), as heurísticas selecionadas garantem melhor resultado para cada movimentação selecionada.

5 Implementação

Nesse capitulo será apresentado mais aprofundadamente as ferramentas e métodos que foram utilizados para realizar os testes e implementações dos modelos de busca de caminho.

5.1 Tecnologias

O projeto é separado em Pathfinder e Pathfinder.UI ambos desenvolvidos na linguagem C# o primeiro utilizando o .NET Standard Library 1.6 e o segundo no .NET Core. No projeto Pathfinder estão todas a implementações dos algoritmos de busca que temos para a comparação. Pathfinder.UI tem como objetivo consumir os recursos do projeto Pathfinder, dando opções de visualização ou geração de dados. Ambos rodam em sistemas Windows e *nix utilizando o .Net Core CLI 1.1 para execução.

5.2 Estrutura do Projeto

Essa seção tem como objetivo descrever como foram implementados os algoritmos

5.2.1 PathFinder

Projeto de implementação de algoritmos de busca. Sua estrutura de pastas está organizada em Abstraction, Core, Factories, Finders, GeneticAlgorithm, Heuristics e MapGenerators.

O **'project.csproj'** é o arquivo onde é definido as bibliotecas utilizadas e a versão do .NET Framework, as outras pastas agregam arquivos com informações relevantes a nossa implementação.

5.2.2 Abstraction

Nesta pasta estão todos os arquivos a nível de abstração dos algoritmos de busca, esses são:

IFactory: Essa interface tem como objetivo padronizar as "fabricas", ferramentas que decidir e instanciar toda dependência necessária.

IMap: Essa interface tem como objetivo abstrair o comportamento da classe de mapa utilizada nos arquivos de busca, assim sendo por padrão todo algoritmo espera uma implementação de IMap para rodar.

IHeuristic: Essa interface abstrai o comportamento das heurísticas.

IMapGenerator: Essa interface tem como objeto abstrair os gerador de mapas.

IFinder: Essa interface é a responsável por abstrair todo comportamento dos algoritmos de busca.

IGeneticAlgorithm: Essa interface herda de IFinder, ela compartilha a mesma assinatura de métodos, propriedades e eventos, porem acrescenta a abstração necessária para o utilização de GA.

5.2.3 Core

Nesta pasta são definidos as implementações e configurações bases.

Container: Esta classe é responsável por registar e resolver as implementações conhecidas das interfaces.

Enumerators: Contem as definições de enumerações, usados para usar nomes bem definidos ao invés de números avulsos no código.

Extensions: Arquivo com métodos auxiliares de lista para comportamento de uma estrutura de pilha.

FileTools: Classe responsável por toda manipulação de I/O de arquivos

Map: Implementação do IMap, tem como objetivo ser a estrutura de mapa base dos algoritmos de busca.

Node: Classe responsável por ser a representação de uma célula no mapa, ou seja, o mapa é uma matriz de ***"Node"***.

Settings: Contém toda configuração estática do projeto, do qual é carregado de um arquivo Json chamado "appsettings.json"

5.2.4 Factories

Nesta Pasta temos os arquivos responsáveis pelo instanciar as implementações de interfaces.

FinderFactory: Classe responsável por decidir e instanciar uma implementação IFinder.

HeuristicFactory: Classe responsável por decidir e instanciar uma implementação IHeuristic.

MapGeneratorFactory: Classe responsável por decidir e instanciar uma implementação IMapGenerator.

5.2.5 Finders

Nesta pasta temos definidas as implementações de todos os algoritmos de busca de caminho.

AStarFinder: Implementação do algoritmo de busca de caminho A* implementada em cima da interface IFinder.

BestFirstSearchFinder: Implementação do algoritmo de busca de caminho "Best First Search" implementada em cima da interface IFinder.

DijkstraFinder: Implementação do algoritmo de busca de caminho Dijkstra implementada em cima da interface IFinder.

IDAStarFinder: Implementação do algoritmo de busca de caminho IDA* implementada em cima da interface IFinder.

GAFinder: Implementação de um algoritmo genético para busca de caminhos implementada em cima da interface IFinder e IGeneticAlgorithm.

5.2.6 Heuristics

Nesta pasta são definidas as implementações de IHeuristic, responsáveis pelos cálculos de heurística.

Manhattan: Implementação da classe Manhattan implementada em cima da interface IHeuristic responsável por calcular a distancia Manhattam.

Euclidean: Implementação da classe Euclidean implementada em cima da interface IHeuristic responsável por calcular a distancia Euclideana.

Octile: Implementação da classe Octile implementada em cima da interface IHeuristic responsável por calcular a distancia Octile.

Chebyshev: Implementação da classe Chebyshev implementada em cima da interface IHeuristic responsável por calcular a distancia Chebyshev.

5.3 Genetic Algorithm

Nesta pasta são definidos todas as implementações referentes ao algoritmo genético, pela complexidade. do algoritmo ele possui uma estrutura própria de pastas para definições e configurações de injeção de dependência.

5.3.1 Abstraction

Nesta pasta estão todos os arquivos a nível de abstração das etapas do algoritmo genético.

ISelection: Interface é responsável por abstrair os algoritmos de seleção.

IGenome: Interface tem como funcionalidade abstrair a definição de genoma.

IFitness: Interface tem como objetivo abstrair o calculo de fitness.

IMutate: Interface tem como objetivo abstrair os operadores de mutação.

ICrossover: Interface tem como objetivo abstrair os operadores de cruzamento.

IRandom: Interface tem como objetivo abstrair a implementação de geração de números aleatórios.

AbstractMutate: Implementação base para operador de mutação.

AbstractCrossover: Implementação base para operador de cruzamento.

5.3.2 Core

Adaptation: Classe responsável para realizar a adaptação de um indivíduo novo após ser gerado.

Enumerators: Contem as definições de enumerações, usados para usar nomes bem definidos ao invés de números avulsos no código.

GARandom: Implementação responsável por gerar números aleatórios, implementa IRandom.

GASettings: Arquivo responsável por carregar configuração estática de GA, carrega do arquivo "GASettings.json".

Genome: Classe responsável por representar o genoma no algoritmo de GA, implementa a IGenome.

5.3.3 Selection

Nesta pasta estão todas as implementações dos algoritmos de seleção.

SelectionRandom: Implementação de seleção de indivíduos aleatório. **SelectionRouletteWheel**: Implementação de seleção roleta.

5.3.4 Crossover

Nesta pasta estão todas as implementações dos algoritmos de cruzamento.

CrossoverOBX: Implementação do operador de cruzamento OBX.

CrossoverPBX: Implementação do operador de cruzamento PBX.

CrossoverSimple: Implementação do operador de cruzamento simples.

5.3.5 Mutation

Nesta pasta estão todas as implementações dos algoritmos de mutação.

MutateBitwise: Implementação do operador de cruzamento Bitwise.

MutateDIVM: Implementação do operador de cruzamento DIVM.

MutateDM: Implementação do operador de cruzamento DM.

MutateEM: Implementação do operador de cruzamento EM.

MutateIM: Implementação do operador de cruzamento IM.

MutateIVM: Implementação do operador de cruzamento IVM.

MutateSM: Implementação do operador de cruzamento SM.

5.4 Projeto de UI

Foi desenvolvido um projeto com objetivo de consumir a biblioteca de busca de caminhos, e poder visualiza-los.

5.4.1 Abstraction

Nesta pasta estão todos os arquivos a nível de abstração.

IAppMode: Abstração que define de que forma o app ira rodar.

IViewer: Abstração do tipo de visualizador.

5.4.2 AppMode

Pode-se configurar diferentes modos de rodar os algoritmo, nesse pasta estão a implementação das diferentes formas.

SingleRunMode: O programa será executado e rodara uma vez usando as configurações do arquivo estático "appsettings.json"que é lido pela classe UISettings.

DynamicMode: O programa ira perguntar qual algoritmo, heurística, tipo de diagonal, forma de visualização e cada operador do GA antes de rodar.

BatchMode: O software ira rodar N vezes cada algoritmo selecionado no arquivo de configuração, onde N também é definido neste arquivo, ao final ira salvar os resultados e cada mapa numa pasta na raiz do projeto.

5.4.3 Core

Nesta pasta são definidos as implementações e configurações bases.

Enumerators: Contem as definições de enumerações, usados para usar nomes bem definidos ao invés de números avulsos no código.

RegisterConfig: Neste arquivo são configurados os binds do visualizador para injeção de dependência.

Settings: Onde são carregados as configurações estáticas do arquivo "appsettings.json", neste são configurações da forma de visualização e do Batch.

5.4.4 Factories

Nesta Pasta temos os arquivos responsáveis pelo instanciar as implementações de interfaces.

AppModeFactory: Classe responsável por decidir e instanciar uma Implementação de IAppMode.

ViwerFactory: Classe responsável por decidir e instanciar uma Implementação de IViewer.

5.4.5 Viewer

Nesta pasta estão as diferentes formas de exibir os resultados das buscas.

ConsoleViewer: Classe responsável por apresentar a busca de caminhos em ASC no Console da aplicação.

OpenGLViewer: Classe responsável por apresentar a busca em uma janela em OpenGL.

OpenGLWindow: Classe que é utilizada pela OpenGLViewer para mostrar a janela com uma grid que mostra o andamento dos algoritmos.

5.5 Estrutura do GA

A busca utilizando o GA, segue com as operações base de todo GA, que são seleção, cruzamento, adaptação e mutação. Para cada interação é gerada uma população, a função de aptidão é calculada para cada indivíduo da população, desses o indivíduo com o valor mais próximo de zero é selecionado como o melhor.

5.5.1 Função de Aptidão

Nesta pasta estão as implementações das funções fitness.

FitnessHeuristic: Para cada indivíduo da população é calculada uma aptidão com base em uma função heurística (Manhattam, Octile, Euclideana, Chebyshev) previamente

definida, o calculo é feito a partir do ponto final da lista do genoma do indivíduo até o ponto de destino do mapa a soma de todos os resultados é a função de aptidão da população.

FitnessWithCollisionDetection: Para cada indivíduo da população é calculada a função de aptidão idêntica a FitnessHeuristic, porém no processo de adaptação adaptação os caminhos que aumentarem para um caminho inválido, que colidam ou saiam do mapa, são marcados, posteriormente todos os indivíduos marcados são penalizados com a soma de um valor alto para diminuir o valor de sua aptidão na população.

5.5.2 Adaptação

Ela é importante para corrigir possíveis problemas nos indivíduos resultantes dos operadores de cruzamento ou mutação. Quando os cromossomos são reorganizados, o caminho novo que foi gerado pode levar para cima de um bloqueio ou para fora do mapa, então seguindo as direções indicadas no cromossomo, a posição no mapa é recalcula e só adiciona o cromossomo do indivíduo se for uma posição valida ou que não voltam para o mesmo lugar. Para complementar o caminho do indivíduo, uma nova direção valida é adicionada e calculada, fazendo com que cada interação de adaptação, o caminho cresça.

5.5.3 Mutação

Todas as mutações executam se o indivíduo tiver mais do que 3 cromossomos e não afeta o primeiro cromossomo. O primeiro cromossomo é a ligação com o ponto inicial e se for trocado de lugar, o caminho é quebrado. /citeMatBuckland

5.6 Modo Batch

O modo Batch do programa serve para poder gerar os dados necessários para a analise.

Para iniciar o processo do batch é necessário o mudar o *AppMode* no arquivo "appsettings.json"para 2 e iniciar a aplicação Pathfinder.UI.

5.6.1 Configuração

A definição das configurações para o Batch ficam no arquivo "appsettings.json"onde são carregadas na classe UISettings posteriormente na aplicação. As chaves que interessam ao modo batch são:

Width: Tamanho em largura dos mapas.

Height: Tamanho em altura dos mapas.

MinimumPath: Tamanho minimo entre o ponto inicial e final dos mapas.

RandomBlock: Tamanho que sera usado para gerar aleatoriamente uma parte do mapa e se repetira para completar os mapas com padrão.

Batch_map_origin: Define se você quer carregar os mapas previamente gerados ou gerar mapas novos para a execução.

Batch_map_qtd_to_generate: Quantidade de mapas que serão gerados para o teste.

Batch_generate_pattern: Este campo define se os mapas gerados automaticamente serão aleatórios (0) ou se terão algum padrão de repetição (1).

Batch_map_diagonal: Define qual tipo de movimentação diagonal sera usada na geração dos mapas, onde Never(0), OnlyWhenNoObstacles(1), IfAtMostOneObstacle(2), Always(3).

Batch_GATimesToRunPerMap: Define a quantidade de vezes que o GA ira rodar para cada configuração.

Batch_folder: Pasta aonde serão gerados os mapas e arquivo de log do batch.

Batch_list_finders: Uma lista que define quais algorítimos de busca serão rodados no modo batch, as opções são A*(0), BFS(1), IDA*(2), Dijkstra(3), GA(4).

Batch_list_heuristics: Uma lista que define quais heurísticas serão rodados no modo batch, as opções são Manhatam(0), Euclidean(1), Octile(2), Chebyshev(3).

Batch_list_Mutation: Uma lista que define quais operadores de mutação serão utilizados pelo GA no modo batch, as opções são *Exchange(0)*, *Displaced Inversion*(1), *Displacement*(2), *Insertion*(3), *Inversion*(4), *Scramble*(5) e *Bitwise*(6).

Batch_list_Crossover: Uma lista que define quais operadores de cruzamento serão utilizados pelo GA no modo batch, as opções são *Simple*(0), *Order-Based Crossover*(1), *Position-Based Crossover*(2).

Batch_list_Fitness: Uma lista que define quais funções de aptidão serão utilizados pelo GA no modo batch, as opções são Heurística(0), Heurística com penalidade em caminho cíclico(1), Heurística com penalidade em caminhos que colidem(2) e Heurística com penalidade em caminho cíclico e colisão(3).

Batch_list_Selection: Uma lista que define quais funções de seleção serão utilizados pelo GA no modo batch, as opções são Aleatório(0) e *Roulette Wheel Selection*(1).

5.7 Modo Visual

O modo Visual do programa serve para poder rodar um algorítimo em um mapa especifico e acompanhar visualmente como ele se comporta.

Para iniciar o modo visual é necessário o mudar o *AppMode* no arquivo "appsettings.json"para 0 e iniciar a aplicação Pathfinder.UI.

5.7.1 Configuração

A definição das configurações para o modo visual ficam no arquivo "appsettings.json"e "GASettings.json"onde são carregadas na classe UISettings posteriormente na aplicação. As chaves que interessam ao modo visual são:

MapViwer: Define o tipo de visualização que sera utilizada onde as opções são Console(0) e OpenGL(1).

MapOrigin: Define a origem do mapa utilizado no programa, onde as opções são Estático(0), De um arquivo(1), Aleatório(2), Com padrão(3).

Heuristic: Define qual heurística sera rodado no modo visual, as opções são Manhatam(0), Euclidean(1), Octile(2), Chebyshev(3).

AllowDiagonal: Define qual tipo de movimentação diagonal sera usada, onde Never(0), OnlyWhenNoObstacles(1), IfAtMostOneObstacle(2), Always(3).

Algorithm: Define qual algoritmo de busca sera rodar no modo visual, as opções são A*(0), BFS(1), IDA*(2), Dijkstra(3), GA(4).

IDAStarFinderTimeOut: Define em quanto tempo em milissegundos o IDA* ira parar caso não encontre o caminho.

IDATrackRecursion: Define se ira mostrar todos os passos recursivos do IDA* (pode demorar).

RandomSeed: Seed utilizada no gerador aleatório de mapas.

Width: Tamanho em largura do mapa.

Height: Tamanho em altura do mapa.

MinimumPath: Tamanho minimo entre o ponto inicial e final do mapa.

RandomBlock: Tamanho que sera usado para gerar aleatoriamente uma parte do mapa e se repetira para completar o mapa com padrão.

FileToLoad: Arquivo de mapa que sera carregado caso escolhe o MapOrigin de arquivo.

FolderToSaveMaps: Pasta onde serão salvas automaticamente os arquivos de

mapas gerados.

Start: Define o caractere de ponto inicial no modo de visualização console.

End: Define o caractere de ponto final no modo de visualização console.

Empty: Define o caractere de caminho livre no modo de visualização console.

Wall: Define o caractere de parede no modo de visualização console.

Path: Define o caractere de caminho no modo de visualização console.

Closed: Define o caractere de nó fechado no modo de visualização console.

Opened: Define o caractere de nó aberto no modo de visualização console.

OpenGlBlockSize: Tamanho em pixel dos blocos no modo OpenGL.

GenerationLimit: Define o numero máximo de gerações do GA antes de abortar o processo.

MutationRate: Chance de ocorrer mutação em cada individuo de uma nova população.

MutationAlgorithm: Define qual operador de mutação sera utilizados pelo GA no modo visual, as opções são *Exchange(0)*, *Displaced Inversion*(1), *Displacement*(2), *Insertion*(3), *Inversion*(4), *Scramble*(5) e *Bitwise*(6).

CrossoverRate: Chance de ocorrer cruzamento em cada par de indivíduos de uma nova população.

CrossoverAlgorithm: Define qual operador de cruzamento serão utilizados pelo GA no modo visual, as opções são *Simple*(0), *Order-Based Crossover*(1), *Position-Based Crossover*(2).

PopulationSize: Quantidade de indivíduos em cada nova população.

FitnessAlgorithm: Define qual função de aptidão sera utilizado pelo GA no modo visual, as opções são Heurística(0), Heurística com penalidade em caminho cíclico(1), Heurística com penalidade em caminhos que colidem(2) e Heurística com penalidade em caminho cíclico e colisão(3).

SelectionAlgorithm: Define qual função de seleção sera utilizado pelo GA no modo visual, as opções são Aleatório(0) e *Roulette Wheel Selection* (1).

BestSolutionToPick: Quantidade de indivíduos mais bem avaliados a serem colocados em uma nova geração a partir da anteriores.

Penalty: Valor de penalidade usado nas funções de aptidão.

6 Conclusão

6.1 Resultados

6.2 Trabalhos futuros

Referências

ALAOUI, O. F. S. M.; EL-GHAZAWI, T. A parallel genetic algorithm for task mapping on parallel machines. 2000. Citado 3 vezes nas páginas 25, 26 e 27.

BJöRNSSON, Y. et al. Fringe search: beating a* at pathfinding on game maps. In: *In Proceedings of IEEE Symposium on Computational Intelligence and Games*. [S.l.: s.n.], 2005. p. 125–132. Citado na página 10.

BJöRNSSON, Y. et al. Fringe search: beating a* at pathfinding on game maps. In: *In Proceedings of IEEE Symposium on Computational Intelligence and Games*. [S.l.: s.n.], 2005. p. 125–132. Citado na página 13.

BLUM, C.; ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 35, n. 3, p. 268–308, set. 2003. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/937503.937505>. Citado na página 13.

BOTEA, M. M. A. Near optimal hierarchical path-finding. *Journal of Game Development, Vol. 1*, p. 7–28, 2004. Citado 2 vezes nas páginas 5 e 9.

BURCHARDT, H.; SALOMON, R. Implementation of Path Planning using Genetic Algorithms on Mobile Robots. In: IEEE. *IEEE World Congress on Computational Intelligence (WCCI 2006), Congress on Evolutionary Computation (CEC 2006)*. Vancouver, Canada, 2006. p. 1831–1836. Citado na página 6.

CAIN, T. *Practical Optimizations for A**. [S.l.: s.n.], 2002. Citado 2 vezes nas páginas 5 e 11.

CHAMPANDARD, J. N. e A. J. *The Secrets of Parallel Pathfinding on Modern Computer Hardware*. 2010. Disponível em: <https://software.intel.com/en-us/articles/the-secrets-of-parallel-pathfinding-on-modern-computer-hardware>. Acesso em: 29/05/2016. Citado na página 15.

CHANG, J. *Making the Shortest Path Even Quicker*. 2009. Disponível em: <http://research.microsoft.com/en-us/news/features/shortestpath-070709.aspx>. Acesso em: 29/05/2016. Citado na página 15.

DEMYEN, D.; BURO, M. Efficient triangulation-based pathfinding. In: *AAAI*. AAAI Press, 2006. p. 942–947. Disponível em: <http://dblp.uni-trier.de/db/conf/aaai/aaai2006.html#DemyenB06>. Citado na página 13.

D.LOHN SILVANO P. COLOMBANO, G. L. H. t. D. S. J. Parallel genetic algorithm for automated electronic circuit design. 2000. Citado 2 vezes nas páginas 25 e 26.

GRAHAM, H. M. e. S. S. R. Pathfinding in computer games. *The ITB Journal*, v. 4, 2003. Citado 2 vezes nas páginas 10 e 28.

HART, N. J. N. P. E.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4, p. 100–107, 1968. Citado 2 vezes nas páginas 8 e 9.

HOLLAND, J. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, USA: University of Michigan Press, 1975. Disponível em: <http://books.google.com/books?id=YE5RAAAAMAAJ>. Citado na página 15.

KARP, R. Reducibility among combinatorial problems. In: MILLER, R.; THATCHER, J. (Ed.). *Complexity of Computer Computations*. [S.l.]: Plenum Press, 1972. p. 85–103. Citado na página 6.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, v. 27, p. 97–109, 1985. Citado na página 13.

KORF, R. E. Linear-space est-first search:. *Computer Science Department*, 1992. Citado na página 13.

KORF, R. E. Linear-space best-first search. *Artificial Intelligence*, v. 62, n. 1, p. 41 – 78, 1993. ISSN 0004-3702. Disponível em: <http://www.sciencedirect.com/science/article/pii/000437029390045D>. Citado 3 vezes nas páginas 5, 6 e 8.

KORF, R. E. Abstraction, reformulation, and approximation: 4th international symposium, sara 2000 horseshoe bay, usa, july 26–29, 2000 proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. cap. Recent Progress in the Design and Analysis of Admissible Heuristic Functions, p. 45–55. ISBN 978-3-540-44914-0. Disponível em: <http://dx.doi.org/10.1007/3-540-44914-0_3>. Citado na página 10.

LEIGH, S. h. J. L. R.; MILES, C. Using a genetic algorithm to explore a*-like pathfinding algorithms. 2007. Citado na página 28.

LUCAS, D. C. Algoritmos genéticos: uma introdução. Universidade Federal do Rio Grande do Sul, 2002. Disponível em: <http://www.inf.ufrgs.br/~alvares/INF01048IA/ApostilaAlgoritmosGeneticos.pdf>. Citado 2 vezes nas páginas 15 e 19.

MACHADO, A. F. da V. et al. Real time pathfinding with genetic algorithm. In: *2011 Brazilian Symposium on Games and Digital Entertainment, Salvador, Bahia, Brazil, November 7-9, 2011*. [s.n.], 2011. p. 215–221. Disponível em: <http://dx.doi.org/10.1109/SBGAMES.2011.23>. Citado 2 vezes nas páginas 10 e 28.

MILLER, W. Applying parallel programming to path-finding with the a* algorithm. 2010. Citado 3 vezes nas páginas 5, 14 e 15.

MOLE, V. L. D. Algoritmos genéticos – uma abordagem paralela baseada em populações cooperantes. 2002. Citado 3 vezes nas páginas 25, 26 e 27.

MUHLENBEIN, H. Evolution in time and space - the parallel genetic algorithm. 2000. Citado 2 vezes nas páginas 25 e 26.

OLIVEIRA, W. A. de. Algoritmo genético para o problema de rotas de cobertura multiveículo. 2009. Citado 2 vezes nas páginas 20 e 21.

PATEL, A. *A*'s Use of the Heuristic*. 2010. Disponível em: <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>. Acesso em: 29/05/2016. Citado 3 vezes nas páginas 10, 11 e 32.

POLLACK, W. W. M. Solutions of the shortest-route problem-a review. *Operations Research*, INFORMS, v. 8, n. 2, p. 224–230, 1960. ISSN 0030364X, 15265463. Disponível em: <http://www.jstor.org/stable/167205>. Citado na página 5.

PONTEVIA, P. Pathfinding is not a star. *Autodesk, white paper*, p. 1–6, 2008. Citado 2 vezes nas páginas 5 e 13.

RATNER, D.; WARMUTH, M. K. Finding a shortest solution for the n x n extension of the 15-puzzle is intractable. In: KEHLER, T. (Ed.). *AAAI*. [S.l.]: Morgan Kaufmann, 1986. p. 168–172. Citado na página 6.

ROSHANI, M. K. S. R. Parallel genetic algorithm for shortest path routing problem with collaborative neighbors. 2015. Citado na página 28.

RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 2. ed. [S.l.]: Pearson Education, 2003. ISBN 0137903952. Citado 2 vezes nas páginas 5 e 12.

SANTOS, A. F. V. M. e. E. W. G. C. U. O. Pathfinding based on pattern detection using genetic algorithms. *SBC - Proceedings of SBGames*, 2012. Citado 7 vezes nas páginas 5, 7, 10, 12, 22, 24 e 25.

SANTOS, U. O. Best first search com algoritmo genÉtico para otimizaÇÃo de espaÇo em problemas de busca de caminhos. INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO SUDESTE DE MINAS GERAIS CÂMPUS RIO POMBA, 2012. Citado 2 vezes nas páginas 6 e 12.

WANG, J.-Y.; LIN, Y.-B. Game ai: Simulating car racing game by applying pathfinding algorithms. *International Journal of Machine Learning and Computing*, v. 2, 2012. Citado 2 vezes nas páginas 13 e 14.

WANG, K. hsin C.; BOTEA, A. Fast and memory-efficient multi-agent pathfinding. In: *In ICAPS*. [S.l.: s.n.], 2008. p. 380–387. Citado na página 12.

WINSTON, P. H. *Artificial Intelligence (3rd Ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1992. ISBN 0-201-53377-4. Citado na página 8.

XU, X. J. *PathFinding.js*. 2016. Disponível em: <https://github.com/qiao/PathFinding.js>. Acesso em: 08/09/2016. Nenhuma citação no texto.