Bruno Pereira Fornaro

ALGORITMO GENÉTICO

Brasil
Novembro de 2022

Bruno Pereira Fornaro

ALGORITMO GENÉTICO

Trabalho de otimização para ciência de dados a respeito da utilização de algoritmos genéticos para realizar a otimização de NPCs em jogos.

Fundação Getúlio Vargas - RJ
Escola de Matemática Aplicada

Professor: Fernanda Maria Pereira

Brasil
Novembro de 2022

Sumário

1 Algoritmo Genético

O algoritmo genético, como o próprio nome indica, é um algoritmo para otimização que se baseia na teoria da evolução genética de Darwin. Dessa forma, o algoritmo simula, por exemplo, a troca de genes em uma reprodução sexuada, a mutação genética e, talvez o mais importante, a reprodução dos indivíduos mais bem adaptados.

Nesse sentido, o algoritmo é bem simples de ser explicado, pois tem um ciclo bem definido:

- Inicializamos uma população: cada indivíduo (também chamado de cromossomo) tem diversos genes que podem representar coeficientes em funções ou outros parâmetros a serem ajustados pelo algoritmo para realizarmos a otimização. Esse genes são definidos aleatoriamente dentro de uma gama de valores possíveis pré definidos (podendo ser inteiros, reais, ou até mesmo letras ou outra abstração, mas normalmente são número binários);

- Testamos a eficiência dos indivíduos: criamos uma função chamada de "*fitness*"para testar se os indivíduos resolvem bem o problema, de forma que indivíduos com melhor desempenho devem retornar uma "pontuação"mais alta;

- Reproduzimos os indivíduos mais bem adaptados: selecionamos indivíduos dois a dois, prioritariamente os indivíduos com melhores resultados na função de *fitness* (mas sem descartar os demais), e misturamos seus genes gerando novos indivíduos. Isto é, definimos um "ponto de corte"nos genes dos indivíduos e geramos novos indivíduos com "metade"(não necessariamente) dos genes dele sendo de um dos pais e a metade seguinte, do outro (podendo, assim, gerar dois indivíduos, cada um com uma das partes de cada um dos pais);

- Aplicamos a mutação: com uma chance bem baixa, fazemos alguns dos genes dos indivíduos serem alterados para valores gerados aleatoriamente, e

- Voltamos a etapa 2 e continuamos o ciclo quantas vezes desejarmos.

Dessa maneira, conseguimos obter um algoritmo de otimização global, que busca maximizar os resultados da função de *fitness* (até mesmo pela natureza do algoritmo, que tenta imitar a natureza). Isso pode ser utilizado para diversas aplicações, desde minimizar funções (podemos manipular levemente a função de *fitness* para obter os coeficientes desejados que minimizam uma função, multiplicando-a por -1, por exemplo) até mesmo criar bons NPCs (*Non-Playable Characters*) em jogos, sendo essa última a que devemos explorar neste trabalho.

2 Aplicação em jogos

Tendo em vista como o algoritmo genético funciona, podemos utilizá-lo para diversas aplicações,

como já foi mencionado, inclusive em jogos. Nesse cenário, o que principalmente estaremos interessados em desenvolver serão NPCs que consigam jogar bem, isto é, fazer a "máquina"jogar como se fosse um ser humano, de preferência com performance melhor. Entretanto, isso pode não ser tão simples de implementar sem ter uma boa compreensão de como abstrair os conceitos do algoritmo para esse contexto. Então, vamos começar explicando por um exemplo de aplicação simples, em um jogo de labirinto.

2.1 Labirinto

No contexto de um jogo de labirinto, desejamos que o NPC seja capaz de atravessar o labirinto a partir de um ponto inicial e chegar a um ponto de destino. Dessa forma, a primeira abstração que podemos fazer é sobre os genes de cada indivíduo, onde uma solução simples para o que eles devem significar é a de interpretá-los um a um como a direção do movimento a ser realizado, sequencialmente. Dessa maneira, considerando que o personagem só pode se mover em quatro direções (para a direita, para a esquerda, para cima e para baixo), podemos representá-las como número de 1 a 4, por exemplo, ou até mesmo como pares de número binários (00, 01, 10 e 11). Assim, criando cada cromossomo como uma cadeira genética com tamanho suficiente para os indivíduos percorrerem o caminho desejado, podemos atingir o resultado que queremos. Nesse sentido, o ideal pode ser que os NPCs encontrem o menor caminho possível, mas até que eles atinjam esse resultado, podemos deixar o cromossomo de cada indivíduo grande o suficiente para realizar um caminho 30% maior que o mínimo possível, por exemplo (MANLIO, 2016).

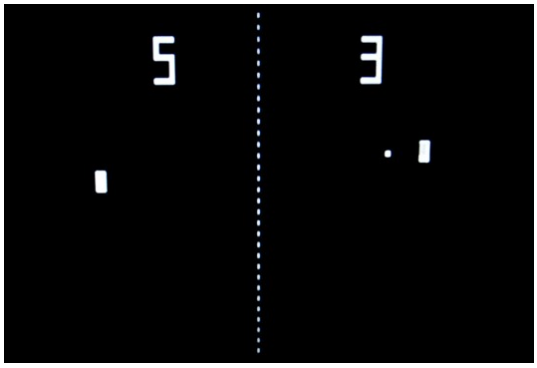
Além disso, outra abstração importante de se atentar é com respeito a como definir nossa função de *fitness*. Uma sugestão é atribuir uma pontuação de *fitness* maior quanto mais perto do ponto de destino o indivíduo terminar ao final da sua vida (isto é, sua pontuação ser inversamente proporcional à distância que chegou do objetivo). Da mesma forma, podemos penalizar na pontuação aqueles indivíduos que passarem duas vezes pelo mesmo local, ou também aumentar a pontuação daqueles que chegarem ao destino com um caminho menor.

De toda forma, vemos que temos certa liberdade para abstrair as funções que precisamos, mas devemos nos atentar a fazer isso de forma correta para que se perpetuem os indivíduos que melhor solucionem o problema desejado. Caso contrário, a evolução pode não convergir para um ponto ótimo, como é esperado.

2.2 Pong (Atari)

Agora que já entendemos num exemplo mais simples como utilizar o algoritmo genético, podemos explicar como utilizamos ele neste trabalho. Dessa maneira, faremos a aplicação no jogo eletrônico chamado Pong (desenvolvido pela Atari, em 1972) (WIKIPEDIA, 2022). Basicamente, o jogo é uma representação 2D de uma partida de tênis de mesa, onde cada jogador é uma raquete (um retângulo em um lado do mapa) e eles devem impedir a bola de passar por eles, rebatendo-a.

Figura 1 – Pong (Atari).



Este exemplo pode ser um pouco mais interessante em comparação ao labirinto, por ter um resultado mais "dinâmico": nosso objetivo será desenvolver um NPC capaz de jogar o Pong e nunca perder. Dessa forma, criaremos um NPC que, depois de treinado, possa ser utilizado para jogar contra humanos e nunca perca, para que alguém possa jogar o jogo "contra a máquina"(sem precisar de outra pessoa para jogar em conjunto).

Neste contexto, começamos recriando o jogo em python. Como a aplicação é bem simples, vamos dispensar a necessidade de colocar o código do jogo aqui, mas é possível encontrá-lo no repositório no GitHub. Entretanto, vamos apenas ressaltar que o campo foi representado como uma matriz 60×100, onde diferentes números inteiros representam o campo vazio, a raquete (com 10 de comprimento) e a bola. Além disso, apenas programamos uma raquete em um lado do campo, dispensando a necessidade de programar a outra raquete em nossa representação, pois nosso NPC apenas precisa ser capaz de sempre rebater a bola, sendo irrelevante a atitude do jogador do outro lado do campo.

Utilizando como base uma implementação de algoritmo genético que já havia sido implementado e publicado em python (GMEA, 2021), criamos cada indivíduo com 9 genes, onde cada um deles representará um coeficiente de uma função de segundo grau no $\mathbb{R}^2$, com valores variando de -10 até 10 (quando gerados pelas funções de inicialização e de mutação), e nossa função de *fitness* sendo:

```
def fitness(population):
    list_fitness = []
    for individual in population:
        game = PongGame()
        count = 0
        while not game.game_over and count < 100000:
            count += 1
            x = game.ball_position[0] - game.player_position
            y = game.ball_position[1]
            w = individual
            move = w[0]*x + w[1]*y + w[2]*x*y + w[3]*x**2 + w[4]*y**2
            move += w[5]*x**2*y + w[6]*x*y**2 + w[7] + w[8]*x**2*y**2
            move = np.sign(move)
            game.play(move)
        fitness = game.score
        list_fitness.append(fitness)
    return list_fitness
```

Explicando melhor, podemos ver que nossa função de *fitness* basicamente utiliza dos genes do indivíduo como os coeficientes (w) de uma função de segundo grau, onde os eixos (x e y) no $\mathbb{R}^2$ são as distâncias relativas da bola com relação a raquete do NPC. Com isso, o que importa do resultado dessa equação é apenas seu sinal, pois ele é utilizado para mover a raquete para uma direção caso o sinal seja positivo (1), para a outra caso seja negativo (-1) ou permanecer parada caso seja 0. A função de *fitness*, dessa forma, mede quantas vezes o NPC consegue rebater a bola ao longo de sua vida, atribuindo um ponto para cada rebatida (quanto mais vezes rebater, melhor). Então, fazemos o indivíduo jogar até que ele perca, ou até que atinja o número máximo de movimentos (definido como 100000), para que o jogo não se torne infinito e evitar que o algoritmo nunca termine.

Além disso, também podemos ressaltar que utilizamos uma função de segundo grau para prever o movimento que a bola irá fazer e deslocar a raquete. Isso pode dar a impressão de que a bola pode descrever uma curva (ou algo nesse sentido), porém o que acontece na realidade é que a bola pode rebater nas paredes, devendo ser considerado esse fator para que a raquete não vá desnecessariamente atrás da bola se ela ainda irá mudar a direção e voltar para próximo da raquete.

3 Resultados

Ao terminar de construir as funções necessárias para utilizar o algoritmo, treinamos ele com populações de 100 indivíduos a cada geração, priorizando os 10 mais bem adaptados a cada geração, com $0,01$ de probabilidade de mutação genética e com 100 gerações.

Os resultados foram muito satisfatórios, pois a pontuação máxima com o número de iterações que limitamos era 507 (pois o campo tem 100 de comprimento mas a raquete ocupa 1) e na primeira geração um indivíduo já atingiu 234 pontos e ao longo das primeiras 4 gerações houve um aumento gradual até chegar em torno de 400 pontos. Sem demorar muito, na geração 22, conseguimos indivíduos atingindo a pontuação máxima.

Após isso, pudemos perceber que a cada nova geração o tempo para testar todos os indivíduos demorava mais. Isso se deve ao fato que os indivíduos começaram a convergir mais de forma que não apenas um se destacava e sim um grupo maior atingia bons resultados, de maneira que todos passavam por quase todas as iterações permitidas. Nesse mesmo sentido, notamos que o algoritmo consome muito poder computacional, pois o tempo para treinar o modelo como descrito mais acima demorou mais de uma hora (embora devemos levar em consideração que o *hardware* utilizado tem peso significativo nisso, e que a implementação foi feita em python, o que impacta muito no desempenho).

Por fim, queríamos testar se o indivíduo desempenhava não apenas com o resultado da pontuação durante a função *fitness*. Dessa forma, pegamos o indivíduo mais bem adaptado (que é salvo depois de treinar o modelo) e colocamos ele para jogar com um limite cem vezes maior de iterações. Assim, o resultado foi que o indivíduo não perdeu e completou todas as iterações, sempre rebatendo a bola, o que reforça que conseguimos criar um NPC que não perde no jogo, como desejávamos desde o início.

Referências

CARVALHO, A. P. de Leon F. de. *Algoritmos Genéticos*. 2009. ⟨https://sites.icmc.usp.br/andre/research/genetic/⟩.

GAD, A. F. *Building a Game-Playing Agent for CoinTex Using the Genetic Algorithm*. 2020. ⟨https://blog.paperspace.com/building-agent-for-cointex-using-genetic-algorithm/⟩.

GMEA. *Como programar algoritmo de Otimização Genética*. 2021. ⟨https://asimov.academy/como-programar-um-algoritmo-de-otimizacao-genetica/⟩. 4

JúNIOR, S. M. B. L. G. T. *Algoritmos Genéticos Aplicados a Jogos Eletrônicos*. 2010. ⟨http://re.granbery.edu.br/artigos/MzU2.pdf⟩.

MANLIO. *How do I implement genetic algorithm on grid board to find optimal path*. 2016. ⟨https://stackoverflow.com/questions/38083172/how-do-i-implement-genetic-algorithm-on-grid-board-to-find-optimal-path⟩. 3

MEDINA, R. M. M. J. *A UTILIZAÇÃO DE ALGORITMOS GENÉTICOS NO DESENVOLVIMENTO DE JOGOS*. 2015. ⟨https://anaisonline.uems.br/index.php/enic/article/view/981⟩.

WIKIPEDIA. *Pong*. 2022. ⟨https://pt.wikipedia.org/wiki/Pong⟩. 3