UNIVERSIDADE D
COIMBRA

Trabalho 2 – Rolling in the Hill

Fundamentos de Inteligência Artificial

FCTUC – Departamento de Eng. Informática 2023/24

Trabalho realizado por:

- Johnny Fernandes PL4 2021190668 uc2021190668@student.uc.pt
- Miguel Leopoldo PL5 2021225940 uc2021225940@student.uc.pt

Índice

Índice de figuras

Introdução

Neste segundo trabalho, foi-nos atribuída a tarefa de completar e aperfeiçoar um algoritmo genético destinado a carros que percorrem ambientes simulados.

O objetivo desta tarefa consiste em desenvolver veículos capazes de se adaptarem e otimizarem o seu desempenho, evoluindo de acordo com as condições do meio onde se encontram, utilizando técnicas ensinadas ao longo deste semestre.

Iniciaremos com a modelação das componentes dos carros, salientando características da implementação que considerámos importantes. Cada aspeto do comportamento adaptativo dos veículos será brevemente explicado.

Modelação e Desenvolvimento

Neste capítulo, exploramos os principais componentes e algoritmos que sustentam o processo de evolução em algoritmos genéticos aplicados à otimização do desempenho dos carros. Através da modelação e desenvolvimento cuidadoso de cada componente - Recombinação, Mutação, Seleção de Progenitores e Avaliação de Aptidão - buscamos melhorar o desempenho dos agentes em direção à meta estabelecida.

Durante a leitura, examinaremos as abordagens e implementações de cada componente, destacando os seus papéis fundamentais na otimização do desempenho de cada carro.

Recombinação

Modelação

A recombinação é realizada através do *Uniform Crossover*, visando que cada descendente contenha genes de ambos os pais. Embora seja improvável, é importante notar que exceções podem ocorrer, nas quais um descendente pode herdar os genes de apenas um dos pais.

Desenvolvimento

O desenvolvimento desta componente foi relativamente simples. Bastou recolher informações sobre o algoritmo e implementá-lo. Inicialmente, os descendentes são equiparados aos pais e, em seguida, entram num ciclo que percorre cada gene, com uma probabilidade de 50% de alterar o gene atual para o do outro pai, de forma espelhada para ambos os descendentes.

```
public IList<IChromosome> UniformCrossover(IList<IChromosome> parents)
{
    // Make copies of the parent chromosomes
    IChromosome parent1 = parents[0];
    IChromosome parent2 = parents[1];
    IChromosome offspring1 = parent1.Clone();
    IChromosome offspring2 = parent2.Clone();

    // For each gene in the chromosomes
    for (int i = 0; i < parent1.Length; i++)
    {
        // Randomly decide whether to swap genes
        if (RandomizationProvider.Current.GetInt(0, 2) == 1)
        {
            // Swap genes between parents
            offspring1.ReplaceGene(i, parent2.GetGene(i));
            offspring2.ReplaceGene(i, parent1.GetGene(i));
        }
    }

    Debug.Log("Uniform Crossover done");
    return new List<IChromosome> { offspring1, offspring2 };
}
```

Figura 1 – Método UniformCrossover

Além deste crossover uniforme, foi também implementado o crossover de k-pontos, que gera um número de pontos de inversão e, em seguida, entra num ciclo que percorre os genes. Quando atinge um ponto de inversão, começa a inverter os genes até chegar a outro ponto de inversão, cessando então a inversão dos genes. Estas inversões podem ocorrer várias vezes. Apesar de ter sido implementado, este crossover de k-pontos não está a ser utilizado, tendo sido implementado apenas para adquirir experiência neste tipo de algoritmos.

```
public IList<IChromosome> KPointCrossover(IList<IChromosome> parents)
{
    // Make copies of the parent chromosomes
    IChromosome parent1 = parents[0];
    IChromosome parent2 = parents[1];
    IChromosome offspring1 = parent1.Clone();
    IChromosome offspring2 = parent2.Clone();

    List<int> switchPoints = new List<int>();

    // Calculate random switch points
    while (switchPoints.Count < KPoints)
    {
        // Generate a random switch point within the chromosome length
        int randomPoint = RandomizationProvider.Current.GetInt(1, parent1.Length - 1);

        // Check if the switch point already exists
        if (!switchPoints.Contains(randomPoint))
            switchPoints.Add(randomPoint);
    }

    // Replace the genes based on the switch points
    bool switchGenes = false;
    for (int i = 0; i < parent1.Length; i++)
    {
        // If the current index is a switch point, toggle the switch state
        if (switchPoints.Contains(i))
            switchGenes = !switchGenes;

        // If the switch state is active, swap genes between parents
        if (switchGenes)
        {
            offspring1.ReplaceGene(i, parent2.GetGene(i));
            offspring2.ReplaceGene(i, parent1.GetGene(i));
        }
    }

    Debug.Log("K Point Crossover done");
    return new List<IChromosome> { offspring1, offspring2 };
}
```

Figura 2 – Método KPointCrossover

Seleção de Progenitores

Modelação

Na escolha dos progenitores, optámos pelo método do torneio. Desta forma, garantimos que os melhores de cada geração se reproduzam, melhorando o desempenho dos agentes. Embora estejamos conscientes das desvantagens deste método, consideramos que, dado o caminho praticamente uniforme, a aleatoriedade que o algoritmo da roleta poderia introduzir não seria vantajosa.

Desenvolvimento

Optamos por uma abordagem intuitiva para a implementação do torneio. Inicialmente, são gerados índices de forma aleatória, entre 0 e o tamanho da população. Em seguida, é criada uma lista de elementos para adicionar os elementos correspondentes a esses índices selecionados. Posteriormente, estes elementos são ordenados de forma descendente e o primeiro é adicionado à lista de futuros pais. Desta maneira, obtemos o melhor elemento do grupo.

```
private IList<IChromosome> TournamentSelection(int number, Generation generation)
{
    IList<CarChromosome> population = generation.Chromosomes.Cast<CarChromosome>().ToList();
    IList<IChromosome> parents = new List<IChromosome>();

    // While the required number of parents is not reached
    while (parents.Count < number)
    {
        // Get unique random indexes of individuals in the population
        int[] randomIndexes = RandomizationProvider.Current.GetUniqueInts(5, 0, population.Count);
        List<CarChromosome> selectedElements = new List<CarChromosome>();

        foreach (int index in randomIndexes)
            // Add the element at the current index to the list
            selectedElements.Add(population[index]);


        // Sort the selected elements by fitness in descending order
        var sortedElements = selectedElements.OrderByDescending(element => element.Fitness).ToList();

        // Add the fittest individual to the parents list
        parents.Add(sortedElements[0]);
    }

    UnityEngine.Debug.Log("Tournament Selection done");
    return parents;
}
```

Figura 3 - Método TournamentSelection

Mutação

Modelação

A mutação pode ser aplicada de diversas maneiras. Nós optamos por uma solução simples, porém eficaz: todos os genes têm uma determinada probabilidade de serem substituídos por outro gene gerado aleatoriamente.

Desenvolvimento

Para a implementação desta componente, foi criado um ciclo que percorre cada gene do cromossoma e, caso seja gerado um número abaixo de uma certa probabilidade, substitui-se o gene por outro gerado aleatoriamente.

```
public void Mutate(IChromosome chromosome, float probability)
{
    // Iterate over each gene in the chromosome
    for (int i = 0; i < chromosome.Length; i++)
    {
        // Check if a mutation should occur for this gene
        if (RandomizationProvider.Current.GetDouble() < probability)
        {
            // Mutate the gene - replace it with a new random value
            chromosome.ReplaceGene(i, new Gene(RandomizationProvider.Current.GetDouble()));
            UnityEngine.Debug.Log("Mutated");
        }
    }
}
```

Figura 4 - Método Mutate

Aptidão

Modelação

Considerando que o objetivo de cada carro é atingir a meta o mais rapidamente possível, a aptidão foi modelada com esse propósito em mente. Assim, a aptidão de um carro é determinada pela distância percorrida em relação ao tempo decorrido. A fim de incentivar os carros a alcançarem a meta, aqueles que não conseguem completar o percurso têm a sua aptidão reduzida em 25%. Dessa forma, os carros evoluem com o objetivo de alcançar a meta o mais rapidamente possível.

Desenvolvimento

Como o desenvolvimento da aptidão baseia-se apenas em cálculos, o processo de desenvolvimento foi praticamente nulo. O cálculo é feito conforme indicado acima, e depois é feita uma avaliação se o carro alcançou ou não a meta. Se não o fez, o resultado do cálculo anterior é multiplicado por 0.75.

```
fitness = Distance / EllapsedTime;
if (RoadCompleted == 0)
{
    fitness = (float)(fitness * 0.75);
}
```

Figura 5 - Cálculo de fitness

Conclusão

Nesta meta, explorámos os principais componentes e algoritmos para otimizar o desempenho dos carros em ambientes simulados. Este é um passo intermédio no processo. Na próxima etapa, realizaremos experiências para testar e validar os algoritmos desenvolvidos, verificando quais valores podem refinar o processo de evolução.