Trabalho 2 - Inteligência Artificial

Isabella Leal Becker[*], Leticia Brasil Flores[†]
Escola Politécnica — PUCRS

22 de novembro de 2023

Resumo

Este relatório apresenta a solução proposta para o segundo trabalho da disciplina, que envolve a implementação de um algoritmo de busca destinado a encontrar um caminho de saída de um labirinto. O algoritmo selecionado para a execução do trabalho é o algoritmo genético. Os detalhes do programa e a análise dos resultados encontram-se detalhados neste documento.

1 Descrição do problema

Este trabalho envolve a criação de programa que deverá fazer a leitura de um arquivo contendo as informações do labirinto, sendo exemplificado abaixo:

Figura 1:

```
E00000000000
111100000111
100001110110
101111110000
000100001011
110001010011
111011000110
001001010110
000011000110
111010011110
111010000111
111000010005
```

Onde "0"marca os caminhos livres para atravessar, o "1"são as paredes, e "E"e "S"são a entrada e a saída, respectivamente. Os movimentos possíveis são: para cima, para baixo, esquerda e direita.

Para atingir o objetivo do trabalho, dividimos em seguintes etapas:

1. Leitura do arquivo texto;
2. Aplicação do algoritmo genético
3. Apresentação do resultado, caminho percorrido no terminal

As próximas seções detalham de forma completa as ações realizadas em cada uma dessas etapas.

[*]isabella.becker@edu.pucrs.br
[†]leticia.flores@edu.pucrs.br

2 Detalhes da implementação

O programa foi desenvolvido em linguagem Java e o código fonte está disponível no repositório no GitHub:

https://github.com/lbflores/T2IA

2.1 Leitura do arquivo de entrada

A leitura do arquivo está implementada na função lerLabirinto, que faz a leitura de um labirinto a partir de um arquivo e retornar uma matriz de caracteres representando o layout do labirinto. O labirinto é lido linha por linha do arquivo, onde cada caractere na linha é atribuído a uma posição na matriz. Os espaços livres são representados pelo caractere "0", as paredes pelo caractere "1", a entrada pelo caractere "2", e a saída pelo caractere "3".

Pseudocódigo função lerLabirinto:

```
1  função lerLabirinto(nomeArquivo):
2      labirinto = matriz de caracteres de tamanho 12x12
3
4      faça:
5          abrir o arquivo para leitura
6          para cada linha no arquivo:
7              dividir a linha em valores
8              para cada valor:
9                  se o valor for "E" ou "S":
10                     atribuir '2' se for "E" ou '3' se for "S" na posição correspondente da matriz
11                 senão:
12                     atribuir 0 ou 1 na posição correspondente da matriz
13     retornar labirinto
```

Também foram definidas a PopulaçãoAtual e PopulaçãoIntermediaria, de 5 linhas de indivíduos e levando em consideração os espaços livres do labirinto, ou seja 0, para determinar a quantidade de movimentos de cada indivíduo gerado.

2.2 Implementação do ciclo do algoritmo de busca

Para implementação do algoritmo genético, buscamos seguir todas as etapas do fluxo do algoritmo, sendo:

```
1
2  1. Gerar uma população inicial
3  2. Avaliar a população
4
5  Enquanto critério de parada não for atingido:
6
7      3. Selecionar os indivíduos mais aptos
8      4. Cruzar os indivíduos selecionados
9      5. Mutar os indivíduos cruzados
10     6. Avaliar a nova população
11     7. Substituir a população antiga pela nova
12     8. Voltar ao passo 3
```

A população inicial foi gerada randomicamente, com valores de 0 a 3, e armazenada na PopulaçãoAtual. Sendo 0 a 3 a codificação dos possíveis movimentos:

0 - para cima
1 - para baixo
2 - esquerda
3 - direita

Logo após selecionar a PopulaçãoAtual é feita uma avaliação da pontuação dessa população, com objetivo de obter a qualidade dos movimentos gerados aleatoriamente em percorrer caminhos válidos no labirinto. A função responsável por essa avaliação é avaliaPopulação, que irá executar para cada indivíduo da população a função heurística.

A função heurística, é a responsável por direcionar o algoritmo genético na criação de indivíduos melhores na busca da solução para o problema, por isso ela é muito importante e foi nesse ponto onde tivemos mais dificuldade, pois sabemos que se a função não estiver boa, não conseguirá direcionar para convergir em uma solução nos próximos passos.

Basicamente a função heurística avalia uma população de movimentos no labirinto, atribuindo uma pontuação com base nos critério de movimentação. Inicialmente, a posição inicial (entrada) é identificada no labirinto, e um caminho percorrido é registrado. Em seguida, a função percorre a população de movimentos, atualizando a posição de acordo com cada movimento e avaliando condições como saída do labirinto, colisões com paredes e repetição de posições. Para essas condições foram atribuídos pesos distintos.

Pseudocódigo função funcHeuristica:

```
1  função funcHeuristica(populacaoAtual , labirinto):
2      pontuacao = 0
3      posicaoAtualX = −1
4      posicaoAtualY = −1
5      caminhoPercorrido = [Posicao(0 , 0)]
6
7      // Encontrar posição inicial (entrada)
8      para cada linha no labirinto:
9          para cada coluna no labirinto:
10             se labirinto[linha][coluna] == '2':
11                 posicaoAtualX = linha
12                 posicaoAtualY = coluna
13                 quebrar
14
15     // Percorrer o caminho e atualizar a pontuação
16     para cada movimento na populacaoAtual:
17         caso movimento seja 0: // Cima
18             posicaoAtualX−−
19         caso movimento seja 1: // Baixo
20             posicaoAtualX++
21         caso movimento seja 2: // Esquerda
22             posicaoAtualY−−
23         caso movimento seja 3: // Direita
24             posicaoAtualY++
25
26         // Verificar condições e ajustar pontuação
27         se posição está fora do labirinto:
28             pontuacao += PEN_SAIU
```

```
            retornar pontuacao
        senão , se colidiu com parede:
            pontuacao += PEN_COLIDIU
            retornar pontuacao
        senão , se alcançou a saída:
            pontuacao = 100000
            retornar pontuacao
        senão:
            para cada posição no caminhoPercorrido:
                se posição.linha == posicaoAtualX e posição.coluna == posicaoAtualY:
                    pontuacao += PEN_REPETIU
            adicionar Posicao(posicaoAtualX , posicaoAtualY) ao caminhoPercorrido
            pontuacao += PON_ACERTOU

    retornar pontuacao
```

Feita a etapa de inicialização da população, passamos para a parte de seleção. Nessa etapa utilizamos os métodos de Elitismo e Torneio.

Primeiramente aplicamos o elistismo, selecionando o individuo com melhor aptidão a colocando na primeira posição da PopulaçãoIntermediária. Em seguida realizamos o torneiro para selecionar individos que representem Pai e Mãe e gerar dois filhos com base no cruzamento dos movimentos de cada um.

O cruzamento, ou crossover, realiza a operação de cruzamento genético entre dois indivíduos representados pelos vetores pai e mae, considerando um ponto de corte determinado pelo labirinto. O código utiliza uma máscara binária para determinar quais genes são herdados de cada um dos pais, gerando dois filhos. O ponto de corte é escolhido a partir da função buscaPontoDeCorte. O primeiro filho (filho1) herda genes do pai até o ponto de corte e, a partir daí, utiliza a máscara para herdar genes do pai ou da mãe. O segundo filho (filho2) segue uma lógica semelhante, mas com a máscara invertida. O resultado final é um vetor contendo os dois filhos gerados.

Pseudocódigo função crossover:

```
função crossover(pai , mae , labirinto):
    corte = buscaPontoDeCorte(pai , labirinto)
    corteMae = buscaPontoDeCorte(mae , labirinto)

    melhorCorte = copiarArray(pai)
    se corte < corteMae:
        corte = corteMae
        melhorCorte = copiarArray(mae)

    tamanho = tamanho do pai
    mascaraBinaria = array de tamanho com valores aleatórios 0 ou 1

    // Gerar primeiro filho
    filho1 = array de tamanho
    para i de 0 até tamanho −2:
        se i < corte , filho1[i] = melhorCorte[i]
        senão , filho1[i] = (mascaraBinaria[i] == 1) ? pai[i] : mae[i]

    // Gerar segundo filho
    filho2 = array de tamanho
    para i de 0 até tamanho −2:
```

```
22          se i < corte , filho2[i] = melhorCorte[i]
23          senão , filho2[i] = (mascaraBinaria[i] == 0) ? pai[i] : mae[i]
24
25      retornar array contendo filho1 e filho2
```

No próximo passo realizamos a mutação, que é a função que realiza alterações nos movimentos de um indivíduo. Primeiro, a função encontra um ponto de mudança e calcula quantos movimentos podem ser alterados depois desse ponto. Em seguida, determina a quantidade total de mudanças, incluindo tanto movimentos válidos quanto os já percorridos. Em loop a função escolhe aleatoriamente movimentos para serem mudados e os substitui por novos valores aleatórios. O processo é feito nas duas partes com base no ponto de corte.

Pseudocódigo função mutacao:

```
1  função mutacao(linhaAtual , labirinto):
2      random = objeto Random
3      tentativasMaximas = 10
4
5      corte = buscaPontoDeCorte(linhaAtual , labirinto)
6      dispovinelParaMutar = tamanho de linhaAtual − corte
7      quantidadeMutacao = (dispovinelParaMutar ∗ 0.5) arredondado para inteiro
8      quantidadeMutacaoParteValida = (corte ∗ 0.10) arredondado para inteiro
9      count = 0
10
11     melhorMutacao = copiarArray(linhaAtual)
12     melhorPontuacao = funcHeuristica(linhaAtual , labirinto)
13     imprime "Melhor pontuação antes da mutação: " + melhorPontuacao
14
15     mutacao = copiarArray(linhaAtual)
16
17     se quantidadeMutacao > 0:
18         enquanto count ⩽ quantidadeMutacao:
19             indiceMovimentoMutado = random.nextInt(corte , tamanho de mutacao)
20             mutacao[indiceMovimentoMutado] = random.nextInt(4) // 0 a 3
21             count++
22
23     count = 0
24     se quantidadeMutacaoParteValida > 0:
25         enquanto count ⩽ quantidadeMutacaoParteValida:
26             indiceMovimentoMutado = random.nextInt(0 , corte)
27             mutacao[indiceMovimentoMutado] = random.nextInt(4) // 0 a 3
28             count++
29
30     melhorMutacao = copiarArray(mutacao)
31     imprime "Melhor pontuação após a mutação: " + melhorPontuacao
32     retornar melhorMutacao
```

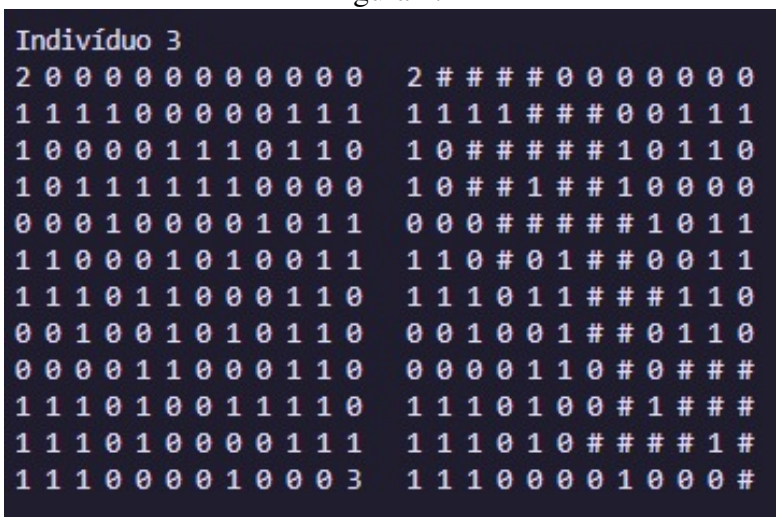
Concluída a etapa de mutação, o último passo é atualizar a populaçãoIntermediária e definir essa como sendo a populaçãoAtual.

Essas etapa citadas ocorrem dentro de um loop no algoritmo, que tem como condição de parada a descoberta de uma caminho que leva a saída ou o atingimento de um número máximo de iterações.

2.3 Apresentação dos resultados

Para demonstrar o melhor caminho encontrado aplicado ao labirinto e possibilitar a visualização do percurso, realizamos a decodificação na função imprimirCaminhoEncontrado, onde é apresentado o labirinto original e o caminho percorrido pela solução sendo representado pelo símbolo "#", conforme figura 2.

Figura 2:



A demonstração é apresentada somente se encontrada uma solução possível, apontando qual indivíduo da população pertence os movimentos.

3 Conclusão

A solução desenvolvida até o momento não converge para a solução em todos os casos testados, que é encontrar uma caminho válido até a saída do labirinto. Conforme testes realizados durante a implementação, identificamos que é necessário otimizar a função heurística para que o programa consiga identificar melhor se a solução é boa ou ruim. Devido ao tempo, não foi possível fazer essas correções, mas o desenvolvimento realizado até aqui foi de grande relevância para entendimento das aplicações e funcionamento do algoritmo genético, bem como da compreensão das etapas necessárias para que o algoritmo de fato aprenda e consiga gerar soluções para o problema com base nesse aprendizado.