Inteligência Artificial

Turma 127 - 2021/2

Profa. Silvia Maria W Moraes

Relatório sobre Algoritmos Genéticos

Carolina Fração, David Bertrand, Lucas Justo e Pâmela Mendonça

1- Forma de codificação

A forma como codificamos o caminho que levará da entrada à saída do labirinto, impactará várias outras partes do algoritmo, como por exemplo detecção de ciclos (é ruim ter ciclos no caminho, por isso é importante detectá-los), onde usar um ArrayList de Point (objeto criado que armazena uma coordenada sendo i = linha e j = coluna) facilita a tarefa, visto que é só procurar por Points que apareçam mais de uma vez no mesmo caminho para achar um ciclo. Já para a tarefa de mutação, por exemplo, se tivéssemos um ArrayList<Point> guardando [(0,0), (0,1), (0,2)] e "mutássemos" para [(0,0), (1,0), (0,2)] ele estaria pulando do (1,0) para o (0,2) sem passar pelas coordenadas entre elas. Visto isso, decidimos usar também uma codificação de ArrayList<Character> onde 'U' = Up, 'D' = Down, 'R' = Right e 'L' = Left, assim facilitando tarefas como mutação e outras. Foram usadas funções de conversão para mudar de uma codificação para outra nos momentos oportunos.

A codificação de coordenadas (ArrayList<Point>) representa um cromossomo (assim como a sua equivalente ArrayList<Character>) que fica armazenado em uma variável ArrayList<ArrayList<Point>> paths, assim formando a população de cada geração. Ao final do ciclo de uma geração, se a resposta não tiver sido encontrada, é atribuída a geração seguinte (gerada com os devidos processos descritos posteriormente neste relatório) na variável paths. Também criamos um vetor de inteiros com o mesmo tamanho de paths, para guardar o valor atribuído pela função de aptidão para cada caminho da geração atual (o valor da função de aptidão para paths.get(i) está armazenado em vetor[i]).

Na hora de apresentar os caminhos em forma de print, foi escolhida a versão de instruções (D, U, L, R). Esta decisão foi tomada por considerar-se mais fácil para um ser humano acompanhar instruções do que coordenadas em um labirinto que possui forma de matriz.

2- Função de aptidão

A função de aptidão gira em torno da variável maxPoints (na qual atribuímos o valor 500, para ter bastante espaço para subtrações). Se o caminho gerado contém a saída

(coordenada (labyrinthSize-1, labyrinthSize-1), neste caso (11,11)) atribuímos maxPoints no valor que a função de aptidão retornará. Se o caminho não possui a saída, atribuímos maxPoints/2 - 5*(distância de Manhattan da última coordenada do caminho para a coordenada da saída). Após isto, sempre temos um valor positivo na função de aptidão e então começamos a subtrair deste valor, baseado em coisas ruins que possam acontecer nos caminhos. O caminho com **maior** pontuação na função de aptidão é o **melhor** caminho, e se a pontuação for igual a maxPoints (neste caso 500) achamos a solução.
Subtrações na função de aptidão:

- -3 pontos para cada parede encontrada no caminho (cada coordenada em que o carácter no labirinto seja '1').
- -5 pontos para cada ponto invalido encontrado no caminho (cada coordenada que tenha i ou j menor que 0 OU i ou j maior que labyrinthSize-1).
- -5 pontos para cada ciclo encontrado (cada vez que uma coordenada estiver repetida naquela caminho, se repetir mais de uma vez subtrai mais de uma vez)

A função de aptidão no código:

```
public static int heuristic(ArrayList<Point> path){
    int points = 0;
    if (containsGoal(path)) {
        points += maxPoints;
    }
    else {
        points += (maxPoints/2) - (5 * manhattanDistanceToGoal(path.get(path.size()-1)));
    }
    points -= 3 * wallsCount(path);
    points -= 5 * invalidPoints(path);
    points -= 5 * cicles(path);
    return points;
}
```

3- Novas gerações

Para criar a primeira população, dentro de um laço, chamamos um método que gera caminhos de maneira aleatória, porém seguindo algumas regras e os adiciona ao ArrayList<ArrayList<Point>> paths. Para gerar esse caminho são seguidas as seguintes regras:

- É adicionado o ponto (0,0) como primeiro ponto do caminho.
- Enquanto o caminho não possui a saída ou enquanto o caminho não atingiu o tamanho máximo é adicionado um novo ponto a ele (tamanho máximo = quantidades de '0' presentes no labirinto).
- Para gerar o próximo ponto, uma função recebe o último ponto do caminho e gera um novo aleatório mas baseando-se nos objetivos de não sair do labirinto (posições menores que 0 ou maiores que labyrinthSize-1) e não passar por pontos já existentes no caminho, o que poderia ficar em loop porque pode ser que chegue em um ponto

que o único jeito de sair seja revisitar um ponto ou sair do labirinto, nestes casos opta-se por revisitar um ponto.

- Depois, ao retornar o caminho gerado para o laço que chamou a função, ele só adiciona este caminho a paths, se não existir nenhum caminho igual.

Desta forma criamos uma primeira população melhor e poupamos tempo ao evitar coisas não permitidas como sair do labirinto. Mas ainda temos que checar por ciclos e saídas do labirinto na função de aptidão porque eles podem acontecer devido ao crossover.

3.1- Critérios de parada

Para saber quando parar de gerar novas gerações, ao final de criá-las faz-se dois testes para saber se não devemos ir para uma próxima. O primeiro é se a **solução já foi encontrada** (se no vetor de pontuação de aptidão tem algum caminho com pontuação = maxPoints) e o segundo é se já geramos o **número máximo de gerações** definido na variável maxGenerations (que pode ser alterada manualmente antes de executar o código). Optamos por **não usar convergência** como método de parada, pois ela é perceptível através dos prints e pode ser evitada ao selecionar um número máximo de gerações menor. Também optamos pela não utilização de convergência para poupar custo de tempo de execução do algoritmo, visto que checar se todos os caminhos estão convergindo antes de gerar uma nova população deixaria o algoritmo mais lento.

3.1- Elitismo

Depois de gerar a primeira geração, chamamos o método que preenche o vetor de pontos de aptidão e escolhemos o maior valor desse vetor para levar o caminho referente a ele para a próxima geração. Assim garantimos que nunca perderemos o nosso melhor cromossomo, pois ele vai sempre ser passado para a próxima geração e protegido contra mutações.

3.2- Cruzamento

Depois de passar o melhor caminho para a próxima geração, ainda temos que gerar mais cromossomos até preencher o tamanho máximo da população. Para isso utilizamos o torneio que escolhe 2 cromossomos aleatórios da geração atual para ver qual deles tem a melhor pontuação de aptidão e ser escolhido para junto a outro cromossomo que foi escolhido pelo mesmo processo, gerar mais 2 cromossomos através do cruzamento uniforme. O cruzamento uniforme vai gerar uma máscara binária e, a partir dela, 2 cromossomos novos com trechos dos 2 cromossomos pais escolhidos por torneio. Para gerar um dos cromossomos, onde na máscara binária há 1 pegamos o trecho do cromossomo pai (escolhido pelo torneio) e onde há 0 pegamos o do cromossomo mãe (escolhida pelo torneio), já para gerar o outro fazemos o contrário, onde na máscara binária há 0 pegamos o trecho do cromossomo pai e onde há 1 pegamos o trecho do cromossomo mãe. Nessa parte

convertemos o caminho para instruções, para não cair no problema das coordenadas se pularem quando misturar as de um caminho com o outro.

3.3- Mutação

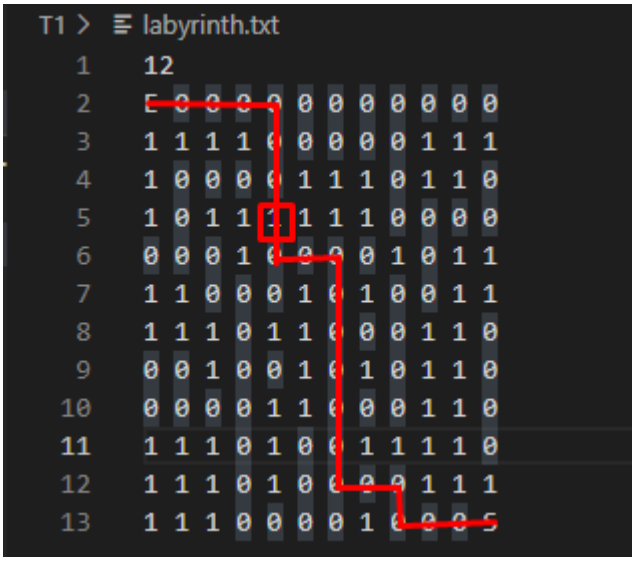
Na mutação também convertemos o caminho para instruções, como explicado na seção 1. Mas antes de converter para instruções, achamos uma parede aleatória (através das coordenadas) e guardamos o index dela. Com o index dela e após a conversão, vemos no index da parede - 1, qual instrução teríamos que ter dado para não cair na parede na próxima coordenada e adicionamos essa instrução no lugar da antiga que levou a uma parede. Essa foi a estratégia final de mutação (mas não a única tentada, como veremos na seção 4) e só é aplicada aleatoriamente para x% cromossomos, com x sendo definido pela variável mutationPercentage (que pode ser alterada manualmente antes de executar o código).

4- Resultado

Com uma taxa de mutação de 50% (alta pois nesse problema precisamos explorar muitos caminhos) e uma população de tamanho 2000. O algoritmo está sempre convergindo para 2 caminhos, na maioria das vezes ele converge para R R R R D D D D R R D D D D D D R R D R R R e poucas vezes para R R R R D D D D R R D D D D D D R R R D R R. O primeiro caminho tem uma parede e o segundo tem duas paredes, e ambos são achados antes da geração 1000 mas levam à uma armadilha que não conseguiram sair nem quando nós executamos para 1 milhão de gerações (demorou alguns minutos).

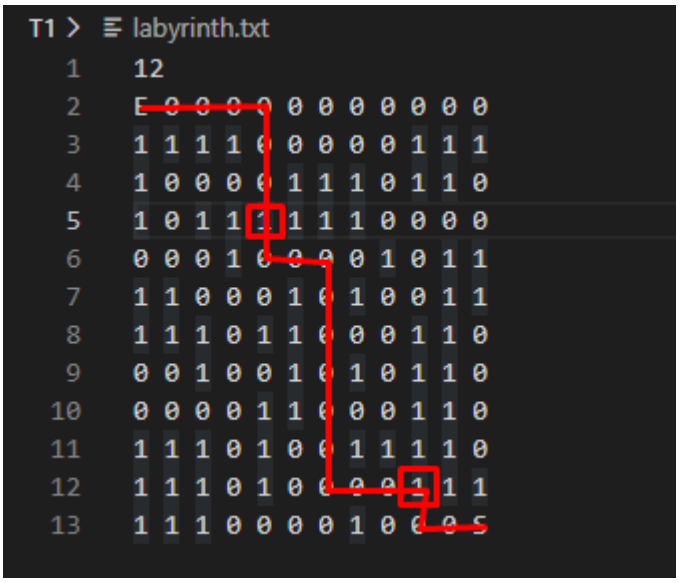
4- Armadilhas nos resultados

Caminho 1:



A armadilha consiste no fato de que se mudarmos a única parede deste caminho, a execução dos passos seguintes levará a um caminho novo com mais de 1 parede, o qual não vai persistir por obter uma pontuação de aptidão pior, assim sumindo das gerações seguintes após os cruzamentos.

Caminho 2:



O caminho dois cai na mesma situação, se mudarmos uma parede o caminho obtém mais paredes, se mudarmos a outra, o caminho chega a sair do labirinto. O entendimento deste problema me fez tentar alguns workarounds, os quais explicarei na seção 5.

5- Tentativas de sair da armadilha

Primeiro pensamos em mudar a mutação, visto que ao mudar somente de uma parede para uma não-parede poderia levar a esse tipo de armadilha, e por isso começamos a “mutar” em mais de um ponto do cromossomo, assim mudando mais de uma instrução, mas isso também não resolveu o problema. Então pensamos que talvez o problema estivesse no torneio, porque se sempre for escolhido somente os melhores para prevalecer, talvez um pior que pudesse sair dessa armadilha no futuro não conseguisse persistir, então também tentamos modificar o torneio para que as vezes escolhesse os piores (com diferentes porcentagens) mas também não resolveu o problema. Também tentamos modificar a função de aptidão, adicionando pontos por quantidade de zeros consecutivos vezes 20 (foram feitas as devidas modificações na condição de parada visto que a solução não seria mais achar algum cromossomo com pontuação de aptidão = maxPoints) para tentar forçar a sair da armadilha, mas não funcionou. Por fim, com falta de tempo, o algoritmo teve que ficar assim (as alterações foram removidas depois de constatado que elas não resolviam o problema). Convergindo para estes 2 caminhos. Os workarounds estão salvos no fim do arquivo Main.java como comentários caso tenha curiosidade.

6- Implementação do A*

Nossa implementação do A* segue os seguintes passos:

1. Atribuir a uma variável que guarda o valor do lugar atual o início do labirinto (0,0);
2. Adicionar a uma lista de visitas pendentes, os lugares vizinhos ao lugar atual;

3. Remover da lista de visitas pendentes duplicatas, locais já visitados, paredes, lugares não visitáveis (maiores ou menores que o labirinto);
4. Ordenar de forma decrescente a lista de visitas pendentes de acordo com a função heurística: F = g + h, onde: F é a função heurística; g é a distância do início **(1 + a distância do anterior)**; e h é a distância do fim (22 - a soma dos valores x e y no labirinto da casa atual);
5. Atribuir à variável do lugar atual o local da lista de visitas pendentes cujo valor da função heurística é menor;
6. Repetir os passos 2-5 até que a variável do lugar atual esteja no fim (11,11);
7. Procurar o caminho do fim até o início de acordo com a diferença da função heurística e atribuir a uma variável;
8. Imprimir essa variável;