山西大学计算机与信息技术学院

**实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓 名 | 李京 | 学 号 | 202202501110 | 专业班级 | 计算机科学与技术2202 | |
| 课程名称 | 人工智能 | | | | 实验日期 | 2024.9.30 |
| 成 绩 |  | 指导教师 | 李琳 | | 批改日期 |  |
| 实验名称 | | 实验二 智能搜索技术 | | | | |
| 一、实验目的：  •通过本实验，学习并掌握智能搜索技术中的两种基础搜索算法：深度优先搜索（DFS）和广度优先搜索（BFS）。此外，学习a-b剪枝技术，理解其在博弈树搜索中的应用。最后，使用遗传算法解决旅行推销员问题，掌握遗传算法的原理和实现过程。   1. 实验内容：   •DFS与BFS搜索算法：实现并比较DFS和BFS两种搜索算法，分别应用于计算给定二维网格中岛屿数量的场景，实验要求使用DFS和BFS各自解决同一个问题。  •α-β剪枝技术：通过博弈树搜索实现α-β剪枝算法，解决博弈树中的极大极小值冗余问题，实验要求构建一个完全二叉树作为博弈树，展示被剪枝的结点。  •遗传算法：利用遗传算法解决TSP问题，通过选择、交叉、变异等操作生成最优解，实验中通过随机初始化城市坐标，计算出遍历所有城市的最短路径。  三、实验平台：  MacBook Air M3  Microsoft Visual Studio Code  Mamba environment + python 3.12.6  四、实验步骤：  DFS & BFS：  遍历网格中的每个单元格。  DFS：当发现一个岛屿（值为'1'）时，调用DFS函数将该岛屿的所有部分标记为'0'（已访问）。  BFS：当发现一个岛屿时，使用队列进行BFS，将该岛屿的所有部分标记为'0'。  计数器增加，最终返回岛屿的数量。  def numIslandsDFS(*grid*):  if not *grid*:  return 0    def dfs(*grid*, *i*, *j*):  if *i* < 0 or *i* >= len(*grid*) or *j* < 0 or *j* >= len(*grid*[0]) or *grid*[*i*][*j*] == '0':  return  *grid*[*i*][*j*] = '0'  dfs(*grid*, *i* + 1, *j*)  dfs(*grid*, *i* - 1, *j*)  dfs(*grid*, *i*, *j* + 1)  dfs(*grid*, *i*, *j* - 1)  count = 0  for i in range(len(*grid*)):  for j in range(len(*grid*[0])):  if *grid*[i][j] == '1':  count += 1  dfs(*grid*, i, j)    return count  grid = [  ["1", "1", "0", "0", "0"],  ["1", "1", "0", "0", "0"],  ["0", "0", "1", "0", "0"],  ["0", "0", "0", "1", "1"]  ]  print(numIslandsDFS(grid)) # Output: 3  from collections import deque  def numIslandsBFS(*grid*):  if not *grid*:  return 0    def bfs(*grid*, *i*, *j*):  queue = deque([(*i*, *j*)])  *grid*[*i*][*j*] = '0'  while queue:  x, y = queue.popleft()  for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  nx, ny = x + dx, y + dy  if 0 <= nx < len(*grid*) and 0 <= ny < len(*grid*[0]) and *grid*[nx][ny] == '1':  *grid*[nx][ny] = '0'  queue.append((nx, ny))    count = 0  for i in range(len(*grid*)):  for j in range(len(*grid*[0])):  if *grid*[i][j] == '1':  count += 1  bfs(*grid*, i, j)    return count  grid = [  ["1", "1", "0", "0", "0"],  ["1", "1", "0", "0", "0"],  ["0", "0", "1", "0", "0"],  ["0", "0", "0", "1", "1"]  ]  print(numIslandsBFS(grid)) # Output: 3  α-β剪枝技术  从给定的层序遍历表示的二叉树中选择最佳节点。它通过遍历树的节点，根据当前层的奇偶性选择最大或最小值的节点，并将剪切值存储在输出列表中。最终，程序打印出所有剪切值。对于大规模的博弈树，Minimax算法的时间复杂度较高，但在简单的二叉树中，Alpha-Beta剪枝可能无法充分利用这一优势，因此采用了简化后的条件判断完成目标。  def minimax(*level\_order*):  nodes = *level\_order*  current = 0  level = 1  outputs = []  while True:  left = 2 \* current + 1  right = 2 \* current + 2  next\_level = level + 1  if left >= len(nodes):  break  if right >= len(nodes):  selected = left  current = selected  level = next\_level  continue  left\_val = nodes[left]  right\_val = nodes[right]  if next\_level % 2 == 1:  if left\_val >= right\_val:  selected = left  cut\_val = right\_val  else:  selected = right  cut\_val = left\_val  else:  if left\_val <= right\_val:  selected = left  cut\_val = right\_val  else:  selected = right  cut\_val = left\_val  outputs.append(cut\_val)  current = selected  level = next\_level  return outputs  if \_\_name\_\_ == "\_\_main\_\_":  input\_str = input().strip()  level\_order = list(map(int, input\_str.split()))  outputs = minimax(level\_order)  if outputs:  print(' '.join(map(str, outputs)))  else:  print('Error')  遗传算法：  城市坐标生成：随机生成10个城市的坐标。  距离矩阵计算：计算城市之间的欧几里得距离，并存储在一个矩阵中。  3. 参数设置：定义遗传算法的参数，如种群大小、代数、交叉率、变异率和精英数量。  种群初始化：生成初始种群，每个个体表示一个城市的访问顺序。  适应度计算：计算每个个体的路径长度，路径长度越短，适应度越高。  选择操作：使用轮盘赌选择法选择父代个体。  7. 交叉操作：使用顺序交叉（OX）生成子代个体。  变异操作：通过交换变异来引入多样性。  9. 精英保留：保留适应度最高的个体，以确保优秀基因传递到下一代。  10. 主遗传算法：在指定的代数内迭代，更新种群并记录最佳路径和适应度变化。  结果输出和可视化：输出最佳路径长度和路径，并绘制城市分布和适应度变化图。  import random  import math  import numpy as np  import matplotlib.pyplot as plt  # Generate coordinates for 10 random cities  NUM\_CITIES = 10  cities = np.random.rand(NUM\_CITIES, 2) \* 100  # Calculate the Euclidean distance matrix between cities  distance\_matrix = np.zeros((NUM\_CITIES, NUM\_CITIES))  for i in range(NUM\_CITIES):  for j in range(NUM\_CITIES):  if i != j:  distance\_matrix[i][j] = math.sqrt((cities[i][0] - cities[j][0]) \*\* 2 + (cities[i][1] - cities[j][1]) \*\* 2)  # Parameter settings  POP\_SIZE = 100 # Population size  GENERATIONS = 500 # Number of generations  CROSSOVER\_RATE = 0.8 # Crossover probability  MUTATION\_RATE = 0.02 # Mutation probability  ELITE\_SIZE = 1 # Number of elites to keep  # Initialize the population  def initialize\_population(*pop\_size*, *num\_cities*):  population = []  for \_ in range(*pop\_size*):  individual = list(range(*num\_cities*))  random.shuffle(individual)  population.append(individual)  return population  # Calculate the path length  def calculate\_fitness(*individual*, *distance\_matrix*):  total\_distance = 0  for i in range(len(*individual*)):  from\_city = *individual*[i]  to\_city = *individual*[(i + 1) % len(*individual*)]  total\_distance += *distance\_matrix*[from\_city][to\_city]  return total\_distance  # Selection operation: Roulette wheel selection  def selection(*population*, *fitness\_scores*):  total\_fitness = sum(*fitness\_scores*)  selection\_probs = [fitness / total\_fitness for fitness in *fitness\_scores*]  selected\_index = np.random.choice(len(*population*), *size*=2, *replace*=False, *p*=selection\_probs)  return [*population*[selected\_index[0]], *population*[selected\_index[1]]]  # Crossover operation: Order crossover (OX)  def crossover(*parent1*, *parent2*):  if random.random() < CROSSOVER\_RATE:  start, end = sorted(random.sample(range(len(*parent1*)), 2))  child\_p1 = *parent1*[start:end]  child\_p2 = [item for item in *parent2* if item not in child\_p1]  child = child\_p2[:start] + child\_p1 + child\_p2[start:]  return child  else:  return *parent1*.copy()  # Mutation operation: Swap mutation  def mutate(*individual*):  for swapped in range(len(*individual*)):  if random.random() < MUTATION\_RATE:  swap\_with = random.randint(0, len(*individual*) - 1)  *individual*[swapped], *individual*[swap\_with] = *individual*[swap\_with], *individual*[swapped]  return *individual*  # Elitism  def elitism(*population*, *fitness\_scores*, *elite\_size*):  sorted\_indices = np.argsort(*fitness\_scores*)  elites = [*population*[i] for i in sorted\_indices[:*elite\_size*]]  return elites  # Main genetic algorithm  def genetic\_algorithm():  population = initialize\_population(POP\_SIZE, NUM\_CITIES)  best\_distance = float('inf')  best\_path = None  history = []  for generation in range(GENERATIONS):  # Calculate fitness  fitness\_scores = [calculate\_fitness(ind, distance\_matrix) for ind in population]    # Record the best individual  min\_distance = min(fitness\_scores)  if min\_distance < best\_distance:  best\_distance = min\_distance  best\_path = population[fitness\_scores.index(min\_distance)]    history.append(best\_distance)    # Elitism  elites = elitism(population, fitness\_scores, ELITE\_SIZE)    # Generate a new population  new\_population = elites.copy()  while len(new\_population) < POP\_SIZE:  parents = selection(population, fitness\_scores)  child = crossover(parents[0], parents[1])  child = mutate(child)  new\_population.append(child)    population = new\_population    return best\_path, best\_distance, history  # Run the genetic algorithm  best\_path, best\_distance, history = genetic\_algorithm()  # Output the results  print("Best path length: {:.2f}".format(best\_distance))  print("Best path: ", best\_path)  # Visualize the results  plt.figure(*figsize*=(10, 5))  # Plot the city distribution  plt.subplot(1, 2, 1)  plt.scatter(cities[:, 0], cities[:, 1], *color*='red')  for i, (x, y) in enumerate(cities):  plt.text(x + 1, y + 1, str(i), *fontsize*=12)  # Plot the best path  path = best\_path + [best\_path[0]]  path\_coords = cities[path]  plt.plot(path\_coords[:, 0], path\_coords[:, 1], *linestyle*='-', *color*='blue')  plt.title('Best Path Illustration')  # Plot the fitness change  plt.subplot(1, 2, 2)  plt.plot(history, *color*='green')  plt.title('Fitness Change')  plt.xlabel('Generation')  plt.ylabel('Path Length')  plt.tight\_layout()  plt.show()  五、实验结果：      六、实验体会：  第一个实验中，DFS和BFS让我体会到递归和队列在处理连通性问题和层级探索中的优势；  Minimax则让我掌握了在树结构中进行决策的复杂性，平衡最大化和最小化策略的挑战。  在遗传算法实验中，我学习了如何模拟自然选择过程，通过适应度评估和选择机制优化解的质量。  这些经历加深了我对算法实现和应用场景的理解。 | | | | | | |
| 教 师  评 语 |  | | | | | |