Report

1

The fitness function is designed to provide a measure of candidate solution quality based on the number of conflicts or attack pairs between queens. The selection strategy is responsible for selecting the best solution from conflicting solutions using the fitness values calculated by the fitness function.

```
def fitness(self):
    conflicts = 0
    for i in range(self.size):
        for j in range(i+1, self.size):
            if (self.positions[i] == self.positions[j]) or (abs(self.positions[i] - self.positions[j]) == abs(i - j)):
                conflicts += 1
    return conflicts
```

To evaluate the fitness of a candidate solution, the fitness function counts the number of invading queen pairs in the solution. A lower number of attacking pairs indicates a better-quality solution. For example, if there are no pairs of invading queens in the solution, the fitness value is set to the maximum possible value. The fitness function works by iterating over all pairs of queens on the board and checking if they are attacking each other. If two queens are in the same row or column, then they are attacking each other. If two queens are on the same diagonal (i.e., the absolute difference in their row and column indices is the same), then they are also attacking each other. We increment a conflict counter each time it finds a pair of attacking queens. Finally, it returns the total number of conflicts found as the fitness value.

```
# Mutates the board by randomly changing one of its positions
def mutate(self, probability):
    if random.random() < probability:
        position_to_change = random.randint(0, self.size-1)
        new_position = random.randint(0, self.size-1)
        self.positions[position_to_change] = new_position
```

In Genetic Algorithm, mutation is the process of randomly changing some aspects of a solution to introduce new genetic material and prevent premature convergence to suboptimal solutions.
We take a mutation probability value and mutates the current solution if a randomly generated number is less than the mutation probability. We first generate a random number between 0 and 1. If the random number is less than the mutation probability, we select a random position on the board and randomly changes its value to a new position.

To be more specific, if the random number is less than the mutation probability, we generate a random integer between 0 and self.size - 1 to determine the position to change. It then generates another random integer between 0 and self.size - 1 to determine the new position to assign to that index in the positions array. Then we set the value of the position at the selected index to the new_position value.

```
def crossover(self, other):
    child_positions = []
    midpoint = random.randint(0, self.size-1)
    child_positions.extend(self.positions[:midpoint])
    child_positions.extend(other.positions[midpoint:])
    child = Board(self.size)
    child.positions = child_positions
    return child
```

Crossover is the process of combining two parent solutions to produce a new child solution. We take two parent solutions, 'self' and 'other', and create a new child solution by randomly selecting a midpoint index and copying the first part of the first parent up to the midpoint, followed by the second part of the second parent from the midpoint to the end. (We first create an empty list to store the child's positions. We then randomly select a midpoint index between 0 and self.size - 1. It copies the first part of the parent 'self' up to the midpoint and append it to the child_positions list. We then copies the second part of the parent 'other' from the midpoint to the end of the positions list and append it to the child_positions list. Finally, we create a new Board object for the child and set its positions attribute to the child_positions list, and then return the new child solution.

The genetic_algorithm function is the main function. It takes in the following parameters:

```
def genetic_algorithm(size, population_size, crossover_probability,
                      mutation_probability, max_generations):
    # Initializes the population with random boards
    population = [Board(size) for i in range(population_size)]
    for generation in range(max_generations):
        # Sorts the population by fitness
        population.sort(key=lambda b: b.fitness())
        # Checking to see if we've found a solution
        if population[0].fitness() == 0:
            return population[0]
        # Selects the parents for the next generation
        parents = population[:population_size//2]
        # Creates the next generation using crossover and mutation
        next_generation = []
        for i in range(population_size - len(parents)):
            parent1 = random.choice(parents)
            parent2 = random.choice(parents)
            child = parent1.crossover(parent2)
            child.mutate(mutation_probability)
            next_generation.append(child)
        # Adds the parents to the next generation
        next_generation.extend(parents)
        population = next_generation
    # The best board we found
    population.sort(key=lambda b: b.fitness())
    return population[0]
```

-size: the size of the board, which is the number of queens to be placed.
-population_size: the size of the population, which is the number of random boards to be generated and used in the genetic algorithm.
-crossover_probability: the probability of performing a crossover operation between two parent boards to create a child board.
-mutation_probability: the probability of mutating a board by randomly changing one of its positions.

-max_generations: the maximum number of generations to run the genetic algorithm.

The function first initializes the population with random boards, and then iterates through a specified number of generations. In each generation, it sorts the population by fitness (the number of conflicts between queens on the board), and checks if a solution has been found (a board with no conflicts). If a solution has been found, it returns the board with zero conflicts.
If a solution has not been found, the function selects the parents for the next generation by taking the top half of the population (based on fitness) as parents. It then creates the next generation by performing crossover and mutation operations on the parents, creating a new set of child boards. The function then adds the parents back to the population and continues to the next generation.
After all generations have been completed, the function sorts the final population by fitness and returns the board with the lowest number of conflicts (the best board found by the genetic algorithm).

The Given solution (solution.poisitions) is a solution to the given size n-queens problem. The function below checks if the number of conflicts or attack pairs between queens of the found solution is zero.

```
def threat_calculate(n):
    # n <-- number of queens
    # if there are < 2 queens
    if(n < 2):
        return 0
    # if there are 2 queens
    if(n == 2):
        return 1
    # if #queens > 2
    return (n - 1) * n / 2
def cost(chess_board):
    dic = {}
    for i in range(len(chess_board)):
      dic[i] = chess_board[i]
    chess_board = dic
    # chess_board <-- a single board of chess

    threat = 0
    m_chessboard = {}
    a_chessboard = {}
    # #queens that are threatening each other diagnolly
    for column in chess_board:
        temp_m = column - chess_board[column]
        temp_a = column + chess_board[column]
        if temp_m not in m_chessboard:
            m_chessboard[temp_m] = 1
        else:
            m_chessboard[temp_m] += 1
        if(temp_a not in a_chessboard):
            a_chessboard[temp_a] = 1
        else:
            a_chessboard[temp_a] += 1
```

```
# Adding the threat cost to threat
for i in m_chessboard:
    threat += threat_calculate(m_chessboard[i])
del m_chessboard
for i in a_chessboard:
    threat += threat_calculate(a_chessboard[i])
del a_chessboard
return threat
```

Which it is:

```
# if the cost is 0 a solution is found
cost(solution.positions)
```

```
0
```

2

We use information from previous generations to guide the mutation, while the random element adds some degree of exploration to the process and helps prevent the algorithm from getting stuck in local optima.

A simple GA algorithm is described as follows:
1. Create a random population of potential solutions consisting of n individuals (initial populations).
2. Evaluate the fitness value f(x) of each individual, x, in the population.
3. Repeat the following three steps to create a new population until completion of the new population.
4. Select two individuals of the current generation for mating.
5. Apply crossover with a certain ratio to create offspring.
6. Apply mutation with a certain ratio.
7. The previous operations are repeated until the completion criterion is met.

Instead of using a single mutation the mutations swap, inversion, dist and distplus were used.

Swap mutation:

```
def swap_mutation(route):
    first_gen, second_gen = random.sample(range(len(route)), 2)
    route[first_gen], route[second_gen] = route[second_gen], route[first_gen]
    return route
```

Inversion mutation:

```
def inversion_mutation(route):
    first_gen, second_gen = random.sample(range(len(route)), 2)
    if(first_gen > second_gen):
        first_gen, second_gen = second_gen, first_gen
    sub_set = route[first_gen + 1 : second_gen]
    sub_set = sub_set[::-1]
    route[first_gen + 1 : second_gen] = sub_set
    return route
```

Dist mutation:

```
def dist_mutation(route):
    first_gen = random.sample(range(len(route)), 1)[0]
    most_close = float("inf")
    for i in range(len(route)):
        if(i != first_gen):
            if(dic_city_length[route[first_gen], route[i]] < most_close):
                most_close = dic_city_length[route[first_gen], route[i]]
                second_gen = i
    temp_number = route[first_gen]
    temp_number_2 = route[second_gen]
    route.remove(temp_number)
    route.insert(route.index(temp_number_2), temp_number)
    return route
```

Dist plus mutation:

```
def distplus_mutation(route):
    route = inversion_mutation(route)
    first_gen = random.sample(range(len(route)), 1)[0]
    most_close = float("inf")
    for i in range(len(route)):
        if(i != first_gen):
            if(dic_city_length[route[first_gen], route[i]] < most_close):
                most_close = dic_city_length[route[first_gen], route[i]]
                second_gen = i
    neiber_second_gen = []
    for i in range(len(route)):
        if(i != second_gen):
            neiber_second_gen.append([route[i],
                                      dic_city_length[route[second_gen], route[i]]])
    neiber_second_gen = sorted(neiber_second_gen, key = lambda x: x[1])
    temp_number = route[first_gen]
    while(True):
        random_gen = random.sample(range(5), 1)[0]
        if(neiber_second_gen[random_gen][0] != temp_number):
            temp_number_2 = neiber_second_gen[random_gen][0]
            break
    route.remove(temp_number)
    route.insert(route.index(temp_number_2), temp_number)
    return route
```

Here all the mutations are added to the next generation.

```
def new_generation(df):
    new_gen_list = []
    for i in list(df['routes']):
        mutation_list = []
        test =  i.copy()
        swap_mut = swap_mutation(test)
        mutation_list.append([swap_mut, calculate_fitness(swap_mut)])
        test =  i.copy()
        inversion_mut = inversion_mutation(test)
        mutation_list.append([inversion_mut, calculate_fitness(inversion_mut)])
        test =  i.copy()
        dist_mut = dist_mutation(test)
        mutation_list.append([dist_mut, calculate_fitness(dist_mut)])
        test =  i.copy()
        distplus_mut = distplus_mutation(test)
        mutation_list.append([distplus_mut, calculate_fitness(distplus_mut)])
        mutation_list = sorted(mutation_list, key = lambda x: x[1],reverse=True)
        for i in mutation_list:
            if(i not in list(df['routes'])):
                new_gen_list.append(i[0])
                break
    total_list = list(df['routes']) + new_gen_list
    df = pd.DataFrame({'routes' : total_list})
    return df
```

The main function:

```
def ga_tsp(df, num_chosens, epoch):
    for i in range(epoch):
        print(i)
        df = new_generation(df)
        df = evaluate(df, num_chosens)
```

The solution for gr229 is:

```
Lenght: 2371.016172677798
```

Path: [87, 94, 100, 101, 34, 38, 95, 96, 75, 91, 36, 37, 35, 106, 107, 117, 97, 98, 92, 93, 102, 103, 74, 76, 77, 84, 20, 16, 17, 18, 23, 24, 83, 82, 81, 85, 25, 86, 89, 56, 57, 55, 58, 54, 53, 59, 61, 65, 64, 66, 63, 62, 60, 67, 78, 79, 90, 80, 88, 73, 71, 72, 70, 68, 69, 229, 228, 227, 217, 218, 224, 225, 226, 52, 51, 7, 5, 4, 6, 9, 8, 3, 10, 11, 19, 15, 14, 26, 28, 29, 32, 39, 40, 41, 181, 180, 179, 177, 50, 193, 191, 49, 188, 43, 44, 45, 46, 184, 199, 159, 150, 123, 122, 120, 121, 119, 118, 115, 116, 113, 108, 99, 104, 105, 112, 109, 110, 124, 125, 166, 167, 169, 170, 164, 165, 173, 174, 172, 163, 178, 182, 187, 198, 197, 176, 175, 200, 204, 212, 205, 206, 207, 208, 209, 210, 216, 215, 214, 213, 220, 219, 223, 190, 189, 192, 194, 195, 196, 222, 221, 211, 203, 202, 201, 158, 157, 156, 160, 161, 162, 127, 126, 111, 114, 138, 130, 128, 129, 137, 136, 153, 154, 155, 151, 147, 145, 146, 149, 148, 152, 141, 143, 144, 142, 140, 139, 133, 171, 131, 132, 135, 134, 168, 183, 185, 186, 48, 47, 42, 31, 30, 33, 27, 2, 1, 12, 13, 21, 22]

The solution for pr1002 is:

```
Lenght: 1579549.426675568
```

Path: [91, 90, 79, 78, 257, 256, 254, 255, 121, 135, 94, 51, 85, 66, 42, 110, 278, 279, 277, 360, 361, 351, 352, 359, 358, 651, 564, 591, 453, 455, 223, 222, 158, 209, 498, 500, 499, 156, 128, 12, 11, 40, 52, 54, 382, 349, 346, 347, 348, 448, 245, 122, 280, 427, 407, 286, 295, 334, 333, 332, 24, 309, 310, 57, 53, 55, 71, 73, 269, 464, 462, 463, 268, 230, 48, 26, 616, 607, 606, 693, 698, 354, 357, 353, 350, 445, 444, 452, 596, 964, 963, 972, 971, 376, 368, 126, 994, 107, 98, 81, 60, 38, 61, 109, 108, 489, 488, 468, 469, 229, 270, 460, 87, 86, 312, 72, 69, 70, 68, 89, 88, 112, 14, 10, 118, 119, 134, 137, 136, 93, 567, 566, 565, 513, 485, 490, 486, 487, 519, 581, 584, 583, 579, 580, 218, 219, 220, 538, 528, 534, 535, 533, 582, 612, 613, 447, 446, 784, 785, 682, 681, 365, 364, 363, 371, 372, 629, 101, 100, 304, 303, 138, 265, 264, 263, 244, 243, 221, 234, 524, 530, 526, 556, 557, 571, 859, 860, 847, 848, 846, 845, 907, 908, 481, 123, 251, 995, 150, 127, 307, 308, 105, 144, 143, 142, 132, 301, 306, 77, 92, 29, 31, 30, 37, 395, 384, 400, 383, 385, 25, 23, 106, 184, 157, 521, 523, 520, 517, 518, 529, 884, 885, 871, 870, 804, 787, 708, 664, 665, 973, 978, 770, 957, 778, 779, 752, 707, 685, 686, 687, 944, 896, 842, 897, 898, 850, 892, 893, 899, 812, 820, 821, 823, 822, 811, 851, 878, 879, 836, 837, 835, 830, 829, 501, 507, 604, 603, 399, 409, 379, 378, 723, 677, 675, 674, 737, 637, 439, 441, 440, 393, 389, 388, 392, 375, 374, 366, 367, 284, 721, 717, 719, 930, 929, 764, 765, 817, 819, 813, 810, 767, 947, 946, 948, 945, 757, 756, 345, 356, 740, 739, 774, 775, 697, 705, 960, 803, 781, 786, 543, 542, 502, 503, 504, 166, 167, 165, 80, 7, 36, 398, 401, 423, 797, 799, 800, 592, 593, 595, 791, 827, 826, 828, 794, 614, 615, 611, 630, 631, 632, 610, 843, 852, 909, 748, 436, 435, 649, 650, 904, 902, 911, 910, 912, 987, 986, 969, 683, 684, 959, 958, 936, 933, 992, 625, 626, 755, 753, 754, 736, 763, 950, 949, 805, 928, 869, 873, 872, 877, 891, 849, 833, 834, 922, 923, 924, 925, 921, 585, 449, 663, 670, 724, 725, 750, 729, 727, 647, 645, 646, 648, 668, 983, 984, 985, 414, 418, 420, 419, 17, 16, 13, 15, 33, 293, 47, 84, 617, 618, 450, 809, 561, 111, 113, 319, 318, 320, 317, 311, 343, 380, 381, 726, 701, 699, 700, 702, 783, 807, 795, 1000, 917, 793, 788, 932, 931, 868, 867, 853, 854, 857, 758, 759, 760, 377, 369, 338, 342, 344, 341, 336, 337, 335, 657, 658, 655, 656, 27, 46, 154, 153, 248, 250, 247, 202, 155, 164, 271, 586, 587, 425, 424, 465, 466, 201, 198, 200, 197, 203, 204, 240, 241, 996, 191, 163, 130, 131, 228, 597, 599, 609, 605, 608, 666, 990, 989, 951, 952, 789, 751, 792, 601, 602, 876, 874, 875, 918, 443, 442, 408, 370, 362, 688, 689, 962, 432, 19, 20, 18, 21, 65, 302, 249, 120, 151, 205, 206, 207, 215, 214, 213, 211, 216, 217, 212, 572, 573, 838, 839, 600, 796, 798, 861, 856, 855, 864, 865, 890, 927, 808, 802, 801, 594, 516, 539, 544, 887, 919, 901, 900, 920, 888, 886, 831, 832, 549, 562, 253, 117, 116, 316, 315, 59, 58, 28, 397, 396, 314, 8, 9, 67, 56, 290, 296, 294, 99, 96, 125, 124, 190, 193, 194, 237, 236, 235, 187, 252, 196, 208, 210, 467, 480, 292, 291, 394, 387, 386, 64, 63, 50, 41, 49, 62, 32, 405, 421, 422, 734, 738, 953, 942, 881, 882, 883, 598, 554, 553, 527, 458, 644, 643, 652, 653, 654, 324, 95, 97, 104, 103, 321, 82, 1, 2, 22, 34, 35, 322, 323, 640, 747, 790, 434, 461, 426, 505, 506, 283, 282, 281, 402, 404, 403, 636, 635, 638, 639, 622, 660, 661, 720, 718, 937, 772, 773, 934, 935, 979, 977, 673, 672, 671, 669, 659, 412, 411, 300, 299, 276, 275, 406, 410, 415, 413, 416, 417, 115, 114, 272, 559, 590, 589, 574, 560, 558, 555, 493, 494, 495, 239, 238, 515, 525, 818, 815, 814, 844, 880, 903, 806, 1001, 965, 966, 967, 968, 954, 993, 961, 980, 981, 982, 976, 974, 769, 766, 477, 472, 474, 475, 476, 340, 339, 313, 39, 633, 634, 438, 997, 642, 690, 694, 704, 703, 761, 762, 746, 741, 749, 451, 454, 509, 508, 457, 456, 473, 186, 199, 195, 548, 540, 545, 840, 816, 841, 895, 894, 695, 709, 745, 735, 744, 742, 743, 722, 715, 714, 706, 696, 941, 940, 956, 955, 938, 620, 619, 623, 621, 624, 728, 712, 975, 733, 905, 906, 926, 916, 889, 824, 825, 563, 437, 355, 691, 692, 676, 943, 939, 776, 915, 914, 716, 662, 532, 531, 537, 536, 224, 233, 496, 497, 173, 171, 172, 159, 161, 226, 227, 225, 510, 999, 511, 541, 782, 780, 732, 731, 730, 913, 777, 627, 628, 285, 45, 44, 43, 149, 147, 146, 305, 991, 328, 327, 329, 330, 331, 182, 183, 231, 140, 141, 139, 152, 242, 148, 145, 433, 491, 492, 484, 483, 471, 470, 258, 259, 428, 430, 429, 998, 478, 479, 273, 274, 641, 679, 678, 680, 713, 711, 710, 667, 988, 970, 771, 768, 373, 390, 391, 289, 288, 133, 129, 102, 185, 176, 175, 177, 170, 76, 75, 74, 6, 3, 5, 4, 326, 325, 168, 169, 160, 162, 181, 180, 178, 179, 174, 246, 261, 262, 232, 575, 576, 588, 188, 189, 192, 267, 266, 260, 512, 522, 550, 552, 551, 546, 547, 577, 578, 1002, 866, 862, 863, 858, 482, 514, 569, 568, 570, 459, 431, 287, 298, 297, 83]