To solve this problem, I decided to use a modified form of Ant Colony Optimization

Ant Colony Optimization is a probabilistic technique for solving computational problem used to approximate a good path between a given set of coordinates, points (which in our case was stars). Ant Colony Optimization uses ants as a pathfinder who drop something called a “pheromone” when going through a trail. The subsequent ant locates the density of pheromone in the path and probabilistically chooses a accurate path.

In my case, I have used Probes which are a representation of ants and paths. The probes always start from Earth – which is the first index in the graph.

I have utilized a Cordinate matrix to keep a track of all 3-D coordinates and made a cordinateMatrix graph which is basically an adjacency matrix that hold values as distance between the two points. Then, I found a good path using ACO and used this to figure what the best path could be from earth.

The timing of the different input depends on a few factors when using Ant Colony Optimization:

* The Number of the probes – The higher the number of probes the more points the probe can explore(probability) and thus more chances to find the shortest distance
* Number of the maximum Iterations: The higher this number, the longer the program will run and try to find a minimal path
* Number of the iteration in each step: The highest this number, the higher the difference between the probability of choosing a trail would be
* Evaporation, density, alpha, beta are all values that can be modified

File and Methods Explanation:

* Probe.java
  + Probe()
    - instantiates Probe class
  + visitStar()
    - tells the probe to visit a star
  + visited()
    - keep a track of visited stars
  + pathSize()
    - distance covered by the probe
  + clear()
    - clear all the visited path - reset
* MainRun.java
  + Contains the main method which reads in user input, the input file, performs Ant Colony Optimization and keeps a track of the time
  + getDistance3D
    - Method used to get distance between two 3D cordinates
* Star\_Path.java
  + public Star\_Path(double[][] cordinatesInput)
    - constructor for Star\_Path that gets input ready for processing
    - sets up all matrix and initializes variables
  + int[] findPath()
    - calls all methods and loops based on the desired input
  + static void setup()
    - set all probes to a start point of earth
  + static void resetMap()
    - reset all the trails – i.e. map to 1 to start fresh
  + static void move()
    - selects a best next path obtained from bestNext() for the probe and moves the probe there
  + static void bestNext(Probe probe)
    - select the bestNext path based on the formulas in probability() and return the value to move()
  + static void probability(Probe probe)
    - Uses the ACO formula to decide between multiple paths. This uses a probabilistic approach where it selects the optimal next path based on the density and probability
    - Formula: density += Math.pow(map[i][l], 1) \* Math.pow(1.0 / cordinates[i][l], 5); : Here 1 and 5 are alpha and beta respectively
      * i is the currentIndex for probe and l is ieration of number of stars
  + static void update()
    - evaporates the density by a certain factor – 0.5 in my case
    - increases the density of a path after a probe has travelled through it
  + static void getBest()
    - selects the best path till date
    - has 2 cases – init and existing – init stores a path if there is nothing there, existing stores a path after comparing it to the old one

The Following result was obtained when running the test for 20, 100 and 1000 stars with distances mentioned below. I have used the first 20, 100 and 1000 subsets of stars since it was easier to work with. Also, I made a new file with just the coordinates for a easy read.

* Setting numberofProbes to 0.8\*number of stars
* Setting attempts = 100
* Setting maxiteration = 100

Enter Number of Stars to Read: 20

Enter Max Distance: 10000

Best tour order: [0, 8, 15, 3, 9, 11, 19, 7, 6, 1, 14, 18, 2, 10, 16, 4, 5, 12, 17, 13]

Possible Path:

0 to 8

8 to 15

15 to 3

3 to 9

9 to 11

11 to 19

19 to 7

7 to 6

6 to 1

1 to 14

14 to 18

18 to 2

Time taken: 168 milliseconds

Enter Number of Stars to Read: 100

Enter Max Distance: 10000

Best tour order: [0, 93, 36, 20, 58, 34, 21, 74, 84, 68, 96, 1, 54, 31, 23, 7, 81, 28, 88, 56, 6, 14, 18, 2, 80, 72, 44, 33, 10, 27, 98, 55, 45, 25, 94, 50, 65, 38, 42, 71, 8, 69, 70, 49, 43, 63, 61, 19, 79, 30, 11, 76, 78, 9, 86, 73, 32, 15, 26, 39, 13, 53, 82, 40, 3, 47, 87, 35, 91, 95, 57, 24, 90, 16, 4, 77, 66, 64, 85, 97, 92, 60, 5, 12, 62, 52, 22, 37, 29, 46, 48, 75, 17, 67, 89, 83, 99, 51, 41, 59]

Possible Path:

0 to 93

93 to 36

36 to 20

20 to 58

58 to 34

34 to 21

21 to 74

74 to 84

84 to 68

68 to 96

96 to 1

1 to 54

Time taken: 3005 milliseconds

Enter Number of Stars to Read: 1000

Enter Max Distance: 10000

Enter Number of Stars to Read: 1000

Enter Max Distance: 10000

Matrix is setup

Best tour order: [0, 908, 514, 991, 820, 141, 502, 412, 61, 590, 167, 619, 103, 937, 958, 769, 471, 101, 166, 88, 417, 14, 339, 756, 617, 877, 409, 939, 653, 60, 917, 812, 672, 536, 946, 533, 919, 605, 375, 698, 524, 410, 733, 539, 633, 614, 762, 142, 460, 567, 456, 645, 75, 557, 563, 717, 903, 806, 69, 538, 623, 298, 720, 876, 784, 382, 813, 926, 707, 690, 728, 469, 643, 716, 776, 844, 44, 521, 72, 329, 210, 553, 238, 845, 785, 383, 648, 237, 513, 299, 527, 718, 584, 703, 158, 679, 963, 704, 626, 857, 269, 461, 555, 323, 906, 775, 724, 453, 833, 677, 880, 83, 493, 546, 218, 370, 487, 436, 205, 647, 105, 575, 900, 870, 881, 200, 373, 52, 956, 458, 827, 65, 143, 872, 572, 673, 415, 190, 144, 898, 459, 229, 582, 509, 865, 185, 676, 744, 31, 580, 866, 348, 496, 609, 692, 606, 431, 392, 706, 385, 198, 625, 497, 175, 699, 832, 137, 189, 693, 259, 281, 354, 233, 181, 889, 659, 257, 747, 787, 202, 779, 818, 91, 110, 795, 751, 283, 488, 603, 384, 663, 515, 830, 263, 868, 416, 224, 798, 887, 587, 821, 134, 801, 736, 790, 180, 494, 491, 368, 314, 258, 577, 726, 277, 849, 598, 350, 632, 525, 78, 882, 560, 293, 519, 701, 681, 261, 743, 664, 964, 551, 242, 220, 297, 649, 758, 737, 650, 163, 67, 295, 62, 361, 500, 380, 146, 512, 99, 172, 124, 547, 352, 136, 443, 678, 631, 616, 969, 753, 76, 408, 777, 541, 574, 231, 22, 611, 287, 3, 362, 222, 740, 207, 662, 177, 959, 94, 97, 861, 4, 203, 19, 430, 765, 864, 106, 855, 176, 965, 308, 121, 49, 447, 972, 333, 688, 540, 253, 462, 463, 781, 46, 713, 739, 470, 755, 178, 173, 639, 658, 932, 884, 503, 695, 814, 119, 59, 549, 376, 290, 492, 485, 518, 397, 216, 38, 395, 346, 419, 25, 364, 192, 481, 624, 344, 390, 426, 252, 406, 188, 794, 708, 400, 997, 592, 526, 156, 987, 33, 945, 934, 508, 799, 709, 407, 321, 24, 342, 628, 960, 528, 935, 50, 303, 82, 885, 236, 838, 520, 441, 81, 568, 18, 159, 490, 304, 42, 479, 445, 771, 405, 240, 335, 687, 228, 246, 147, 862, 612, 306, 127, 517, 123, 377, 309, 149, 128, 629, 454, 230, 651, 748, 100, 746, 967, 984, 689, 114, 878, 714, 804, 194, 171, 184, 422, 32, 366, 73, 86, 9, 164, 970, 302, 438, 916, 976, 74, 21, 247, 475, 40, 401, 336, 921, 421, 379, 618, 637, 227, 195, 8, 841, 48, 51, 556, 561, 245, 792, 95, 763, 962, 920, 933, 912, 111, 752, 952, 117, 358, 288, 391, 473, 507, 819, 211, 732, 120, 165, 108, 266, 341, 13, 251, 542, 226, 719, 809, 139, 613, 836, 15, 291, 566, 433, 427, 1, 402, 393, 169, 154, 187, 615, 816, 118, 948, 974, 904, 356, 112, 786, 634, 890, 923, 641, 145, 267, 668, 929, 750, 338, 208, 209, 20, 735, 301, 70, 324, 225, 413, 135, 93, 764, 510, 966, 442, 895, 56, 680, 282, 271, 133, 28, 41, 386, 548, 591, 363, 168, 337, 87, 418, 982, 869, 250, 451, 826, 204, 153, 151, 999, 817, 357, 160, 604, 85, 107, 387, 464, 793, 824, 381, 234, 89, 622, 858, 197, 924, 347, 943, 126, 359, 320, 327, 585, 570, 981, 45, 398, 940, 80, 902, 435, 670, 179, 522, 559, 355, 388, 109, 854, 276, 196, 404, 915, 64, 29, 774, 851, 599, 620, 37, 132, 828, 696, 68, 239, 219, 389, 305, 26, 636, 998, 300, 489, 294, 472, 36, 152, 954, 316, 217, 957, 850, 66, 661, 396, 466, 315, 942, 260, 910, 193, 125, 349, 992, 353, 92, 516, 573, 607, 913, 936, 104, 760, 930, 307, 973, 860, 788, 2, 157, 6, 896, 627, 564, 767, 280, 581, 322, 270, 823, 174, 285, 273, 506, 244, 667, 501, 484, 34, 576, 206, 58, 674, 738, 691, 950, 249, 241, 140, 511, 77, 534, 990, 842, 684, 455, 734, 961, 586, 754, 928, 802, 630, 328, 796, 829, 843, 656, 545, 439, 768, 642, 700, 130, 552, 5, 317, 467, 837, 911, 71, 326, 450, 399, 655, 846, 725, 979, 486, 262, 424, 374, 944, 311, 254, 131, 213, 978, 710, 43, 749, 495, 214, 666, 597, 449, 63, 150, 170, 90, 808, 313, 11, 544, 17, 499, 284, 113, 558, 47, 847, 420, 770, 351, 523, 57, 403, 191, 608, 39, 644, 161, 980, 182, 292, 215, 369, 853, 27, 989, 476, 831, 10, 434, 116, 440, 265, 863, 429, 478, 186, 697, 730, 7, 23, 148, 805, 994, 811, 248, 731, 54, 757, 129, 685, 543, 537, 715, 428, 941, 96, 550, 931, 867, 201, 554, 894, 905, 907, 84, 918, 766, 394, 602, 657, 600, 223, 378, 914, 727, 721, 325, 474, 789, 852, 893, 640, 871, 569, 953, 296, 452, 343, 583, 457, 635, 35, 759, 255, 922, 638, 977, 138, 594, 971, 722, 951, 686, 983, 927, 601, 654, 340, 345, 423, 310, 825, 873, 122, 98, 55, 448, 886, 675, 595, 183, 275, 729, 621, 646, 705, 579, 652, 480, 671, 332, 16, 840, 985, 891, 232, 444, 879, 212, 807, 102, 859, 773, 272, 593, 12, 803, 331, 897, 694, 901, 665, 988, 531, 711, 783, 446, 682, 532, 372, 365, 115, 477, 289, 780, 318, 468, 995, 772, 243, 610, 425, 742, 660, 968, 683, 367, 815, 925, 53, 848, 888, 371, 741, 975, 498, 414, 702, 278, 875, 835, 669, 947, 235, 797, 810, 839, 791, 565, 883, 530, 955, 482, 822, 588, 571, 589, 155, 596, 745, 360, 162, 892, 256, 874, 993, 778, 856, 712, 30, 264, 79, 221, 562, 199, 529, 535, 465, 996, 274, 437, 432, 800, 578, 909, 483, 268, 761, 938, 949, 782, 312, 330, 319, 279, 504, 334, 834, 286, 505, 986, 723, 411, 899]

Possible Path:

0 to 908

908 to 514

514 to 991

991 to 820

820 to 141

141 to 502

502 to 412

412 to 61

61 to 590

590 to 167

167 to 619

619 to 103

Time taken: 271 milliseconds

This was only possible to run in 271 ms when we had 2 probes working, 1 max iteration and 10 iteration per probe

This talks about how flexible the algorithm is. This path might not be the most efficient path but this is definitely very fast to calculate. We can modify all the parameters in ACO according to our need and our time constraints.

For eg: When using about 800 probes, 100 max iteration and 100 iteration per probe, the algorithm did not produce an output in over 15 minutes. It could have given an output but it would take a long time. I just changed 3 values and I got almost-perfect output.

**What could have gone better?**

* I should have started the project as soon as it was assigned. This would have given me more time to think about alternatives
* The data could have been read from the main file itself rather than making a new file. If I had more time, I would work on this. Also, it would be awesome if I could use random/some heuristic to select good indexes rather than going sequential.
* Ant Colony Optimization is a probabilistic method hence it is not always reliable. However, since our problem was such that it was hard to find an ideal solution, I think ACO does a great job in getting results in a timely manner with a lot of flexibility
* Since my algorithm finds a full path first and then uses it to find a path that the probe can visit, it can take a long time to locate an exact path. But this path can be utilized later as well when the distance grows

In summary, ACO provides a great flexibility in getting correct answer vs time constraint. It can find a near perfect answer in maximum time and mere perfect answer in very less time. If we need to find answers quickly, we can just ierate once with a couple of probes. This gives us a good approximation of the answers