**Solution encoding representation**

For the random sampling solution, as I only had to generate a random list of coordinates based on the possible locations and did not have to take into account the solution that had been used previously, I used a vector of discrete values. To do so, I generated a list made up of random coordinates made up of possible coordinates according to the initial p locations generated, with the only requirement being that there were no repeated coordinates.

Example: [(17300, 79781), (54141, 84174), (2834, 62348), (4348, 30638), (82048, 67415), (78365, 20576), (95711, 37796), (56135, 29607), (33269, 18803), (12088, 42091), (53961, 3791), (48969, 96790), (42609, 49979), (2904, 75558), (90142, 79709)]

For local search and simulated annealing, I used binary encoding to determine which of the locations to build a facility, with “1” representing a facility and “0” representing none. The index of the binary encoded list mapped to the index of the list containing the coordinates of all possible locations. I then tried different solutions by randomly changing the locations of the “1”s and “0”s and calculating the total distance to determine whether to keep the swap.

Example: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0] maps to [(49543, 50472), (22876, 86100), (90142, 79709), (2904, 75558), (23086, 6951), (2148, 27291), (18590, 47377), (99254, 2363), (33269, 18803), (77844, 63094), (57779, 86778), (45913, 97377), (78365, 20576), (88617, 43295), (56135, 29607)]

**Search Procedures and Results**

*Random Search*

For random search, random possible solutions are tested, with the most optimal solution out of all the ones tested being kept.

For my code, a while loop is used to run the code for the set amount of time (30 or 60 seconds). For each loop, a random set of *p* coordinates is generated as explained previously under the solution encoding of random search. Each coordinate with the same index maps to the same point. This random set of coordinates is set as the locations where the facilities are built. A function is then used to find the closest facility to every other location that does not have a facility built, and the distance between that location and the facility is calculated. The total of all the distances is calculated and compared to the previous total. The minimum total distance is kept, along with the respective coordinates of the facilities for the minimum total. Random combinations are tested until the time is up, and the combination of coordinates with the lowest total distance is kept.

Instance 1 best objective value found: 909508.3869843065

[(17300, 79781), (54141, 84174), (2834, 62348), (4348, 30638), (82048, 67415), (78365, 20576), (95711, 37796), (56135, 29607), (33269, 18803), (12088, 42091), (53961, 3791), (48969, 96790), (42609, 49979), (2904, 75558), (90142, 79709)]

|  |  |
| --- | --- |
| Solution Plot | CPU Performance Plot |
| A diagram of a network  Description automatically generated | A graph of a number of individuals  Description automatically generated |

Instance 2 best objective value found: 8123716.654311163

[(70802, 93295), (23837, 53865), (10693, 14085), (88010, 76148), (17300, 68120), (42211, 64506), (71343, 60738), (35449, 83265), (90142, 18120), (68818, 73103), (17236, 89190), (23211, 4584), (60499, 55036), (25258, 70786), (848, 77207), (31365, 61203), (51505, 47062), (95246, 40516), (91314, 87036), (48449, 54202), (84626, 39198), (63373, 28876), (10316, 23396), (77575, 27777), (5534, 42043), (10468, 53226), (33866, 20154), (34244, 62974), (59361, 87437), (62996, 1904)]

|  |  |
| --- | --- |
| Solution Plot | CPU Performance Plot |
| A diagram of a network  Description automatically generated with medium confidence | A graph of a number of objects  Description automatically generated |

*Local Search*

Local search works by introducing small perturbations to the existing solution and accepting the new solution if it is an improvement, else the change is reverted, and other perturbations are attempted. The new solution is kept only if it is an improvement and another perturbation is introduced on the new solution, repeating the cycle to search for a similar solution that is better and slowly improve the solution.

In my code, the same method as before is repeated to track of the time, as well as find the closest facility and calculate the total distance. However, to determine the locations of the facilities, I employed a different encoding method as explained previously. I first start by generating a list with *n* elements, with the first *p* being “1,” indicating the presence of a facility at that location, and the rest of the elements being “0.” I then keep track of the last positions where a “1” is swapped to “0” and vice versa. I introduce the perturbation by moving the pointer on the “1” and “0” by a random number (to prevent the same location from being checked repeatedly) between “1” and “5” until the next “1” and “0”, respectively, is found. The values at these two positions are then switched. This effectively changes the location of one of the facilities, and the total distance is calculated as previously explained. The swap is preserved only if the total distance is smaller than the previous minimum found, else the “1” and “0” are swapped back. The pointers are then moved once again to introduce another perturbation. This continues until the time runs out, with the solution that generated the minimum total distance kept.

This solution has a high likelihood of getting stuck in a local minimum when the situation where more than one facility location has to be changed at once in order to get a better solution is encountered. However, the likelihood of getting a better solution is much higher than when random search is used.

Instance 1 best objective value found: 798170.8605265633

[(49543, 50472), (22876, 86100), (90142, 79709), (2904, 75558), (23086, 6951), (18590, 47377), (99254, 2363), (33269, 18803), (67030, 81703), (77844, 63094), (52093, 5878), (57779, 86778), (78365, 20576), (88617, 43295), (56135, 29607)]

|  |  |
| --- | --- |
| Solution Plot | CPU Performance Plot |
| A diagram of a network  Description automatically generated | A graph with numbers and lines  Description automatically generated |

Instance 2 best objective value found: 6792409.910803573

[(9385, 14885), (3058, 87991), (49581, 49373), (17034, 32182), (25529, 87912), (45470, 95233), (57584, 81970), (88726, 94678), (31328, 39548), (49326, 66175), (94922, 34772), (72989, 55219), (60067, 58272), (62752, 21393), (80865, 6346), (39940, 13325), (26906, 57570), (23211, 4584), (30549, 18298), (91367, 74790), (6080, 40676), (71938, 37054), (74851, 85590), (89279, 13358), (92516, 57603), (38341, 67538), (58957, 3852), (79328, 26207), (11615, 55212), (52074, 28253)]

|  |  |
| --- | --- |
| Solution Plot | CPU Performance Plot |
| A diagram of black dots and lines  Description automatically generated | A graph of a number of objects  Description automatically generated with medium confidence |

*Simulated Annealing*

Simulated Annealing is very similar to local search, with the main difference lying in the probability a solution is accepted if it is not an improvement. With local search, this probability is 0, but with simulated annealing, this probability is calculated using a function that relies on the difference between the previous solution and the current solution and how long the search has been running for. The longer it has been running, the less likely the worse solution will be accepted. In my code, I used the equation

*Probability = e\*\*((prev-curr)/ (initial\_temp\*(alpha\*\*iteration)))\*100*

where *alpha* and *initial\_temp* are arbitrary constants that are calibrated through trial and error (The values I used will be explained below.) *Prev* refers to the total distance of the current best solution, *curr* refers to the total distance of the current solution being testing and *iteration* refers to the iteration the while loop is currently at, as each time a loop is run, the iteration is increased by 1.

When *curr* is smaller than *prev*, the new solution with a smaller total distance is always accepted and kept as the new solution to introduce new perturbations in the same way as it was done in local search. However, when *curr* is larger than *prev*, the probability is calculated based using the equation and a random number between 1 and 100 is generated. If the number is smaller than the probability, the new solution that is worse is accepted and replaces the current solution. Perturbations will then continue to be introduced and the cycle continues. The larger the difference between *curr* and *previous*, and the larger the number of iterations, the lower the probability the worse solution will be accepted. This ensures that there is greater exploration of solutions at the beginning of the search as worse solutions are accepted more often. In contrast, towards the end of the search, the probability of accepting a worse solution is extremely close to zero and only better solutions are accepted. When the run time is up, the best solution is kept.

This method helps to alleviate the problem of getting stuck in a local minimum, although it can still happen towards the end of the search when only better solutions are accepted. There is also the risk of a solution that is much worse being accepted at the beginning without a simple way to get to a better solution, which is why calibration of the arbitrary constants is extremely important.

While testing, I focused on trying different values of *alpha*. For *instance 2*, I tried the cases of alpha = 0.999, 0.9995, 0.9999. These were the CPU Performance Plots obtained.

|  |  |  |
| --- | --- | --- |
| Alpha = 0.999 | Alpha = 0.9995 | Alpha = 0.9999 |
| A graph of a number of objects  Description automatically generated | A graph of a number of objects  Description automatically generated with medium confidence | A graph of a number of objects  Description automatically generated with medium confidence |

From the graphs, it can be seen that a larger value of alpha leads to a longer period of exploration. While this helped to avoid local minimums, accepting worse locations towards the end resulted in solutions that could have been improved on being missed out. In the end, I chose to use 0.9995 as it gave the best balance of exploration and improvement and produced the best results. For *instance 1*, to compensate for the shorter runtime, alpha = 0.999 was used instead to ensure that there was sufficient time spent on improving the solution.

Instance 1 best objective value found: 791736.9605361611

[(49543, 50472), (22876, 86100), (90142, 79709), (2904, 75558), (43788, 10948), (21878, 15962), (2148, 27291), (18590, 47377), (99254, 2363), (77844, 63094), (48969, 96790), (70338, 85851), (78365, 20576), (88617, 43295), (56135, 29607)]

|  |  |
| --- | --- |
| Solution Plot | CPU Performance Plot |
| A diagram of a network  Description automatically generated | A graph with a line  Description automatically generated |

Instance 2 best objective value found: 6771591.72371095

[(9385, 14885), (70304, 7737), (50842, 26933), (39325, 13903), (50223, 94867), (87871, 5566), (88726, 94678), (37803, 66226), (88506, 53675), (65201, 19650), (22506, 9402), (56040, 4265), (87742, 19707), (76602, 87749), (29547, 84714), (31125, 30132), (3053, 43735), (16562, 33815), (71574, 69076), (49236, 57404), (71493, 37075), (91367, 74790), (6884, 68451), (92544, 35599), (58626, 83419), (69132, 56308), (8719, 91177), (50781, 42740), (21462, 61813), (26753, 44338)]

|  |  |
| --- | --- |
| Solution Plot | CPU Performance Plot |
| A diagram of black dots and lines  Description automatically generated | A graph of a number of objects  Description automatically generated with medium confidence |