Both implementations make use of a Matrix class which reads in data from a city-file, clean it before storing it as a 2D array where [0][1] is distance from node 1 to node 2. (Starts from 0).

**Implementation A: Greedy Best First Search**

I started with **Nearest Neighbour** algorithm:

* Start on a random node and set it to visited
* Find shortest edge connecting current node and nearest unvisited node
* Set current node to this nearest node and set it to visited.
* If all nodes are visited, terminate
* Else, go to bullet point 2.

Made use of a stack. Gave often poor results, an upper bound for tour length.

**Improvements**

I then used this nearest neighbour algorithm as the basis for my heuristic in a **Greedy Best First Search Algorithm with Heuristic information. (more involved algorithm)**

Greedy Best first search simply chooses the unvisited node with the best heuristic value to visit next. The priority of each node added to the queue is determined by its heuristic. Defines the evaluation function to be equal to the heuristic function; that is, f(z) = h(z).

**Implementation of Greedy Best First Search**

* Set an arbitrary node as source node.
* Add source node to priority queue.
* Enter a loop, while priority queue is not empty.
* Node to move to is the node at the top of the priority queue i.e. lowest f value. Pop this node off the queue. Set this node to visited.
* Clear priority queue.
* Add the distance between said node and its parent’s node and add it to a variable runningTotal.
* Add the node to an array list ‘order’ to keep track of tour.
* Loop through all of the neighbours of this node and for each of these neighbours obtain their heuristic value. (shown below). If a neighbour is a valid node to move to, (i.e. not moving to itself and not already been visited) add this node and its associated heuristic to the priority queue. If goal node is found, i.e. no neighbours, final move is from goal node back to source node.
* Return to bullet point 4.

Need to clear the priority queue as after each iteration; a nodes heuristic vale depends on what nodes have been visited, i.e. need to be recomputed.

Essential to the priority queue is how one compares nodes. I used a Vertex class which implemented Comparator. Each vertex has an associated node and and heuristic value and each node is compared by its heuristic value. Node with smallest h value given highest priority, i.e. top of the queue.

**Data Structures and data representations**

-Made use of PriorityQueue type <Vertex>

-Only argument was 2D array of cities.

-Kept tour order in an array-list so I could print at end.

-Need to add one to each node in the tour order as arrays go from 0 and I want my first node to be 1.

-visited = new int [numberOfNodes + 1] to keep track of which nodes have been visited.

0 being unvisited, 1 being visited.

**Heuristics**

I started with a heuristic that, given a state t, h(t) is defines to be the minimal step cost in moving to the nearest legal state.

To **improve** I changed it to:

Given some state t, heuristic h(t) is the sum of the distance of the closest city c that has not been visited to the last city of the partial tour t plus the distance of any other unvisited city to the start city.

This heuristic reflects that you want your next city to be visited to be close to the current city but that you don’t want to be left with a city that is a long way from the start city. This gace much better tours.

**Further Improvements**

-Originally Started at Node 1 every time, but introduced a random number generator at the beginning to select a starting node between 1 and number of cities.

-Then changed the program so it completed the algorithm for every node as the source node and then selected the best tour. This gave much better tours as best first search depends on which node you start from.

**Implementation B: Simulated Annealing (more** **involved algorithm**)

The algorithm proceeds by iteratively performing a local search from the current state X until the temperature drops to 0 at which point the algorithm halts. While this temperature variable is high the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local optimums it finds itself in early on in execution. As the temperature is reduced so is the chance of accepting worse solutions, therefore allowing the algorithm to gradually focus in on an area of the search space where a close to optimum solution can be found.

**Process**

Note throughout this process, an ‘Individual’ is a tour through the cities.

In my SimulatedAnnealing class, I set initial temperature and cooling rate, created a currentIndividual and randomised this Individual through generateRandomIndividual() method in my Individual class. This is done by the method: Collections.shuffle(cities).

I set this Individual to the best Individual obtained so far.

While temperate was greater than 0, I entered a loop:

* Generate new individual based on current Individual, i.e. by swapping two nodes around.
* If this change brought about a better (shorter) tour, set current individual to this new altered individual.
* If the change brought about a worse tour, set it to current individual only if the probability (defined below) was greater than a random number generated by Math.random().

(With the other eventuality being that it brings about no change therefore we also set this new individual to be the current individual).

*Probability = exp( (solutionEnergy - neighbourEnergy) / temperature )*

* The ‘cooler’ the temperature T, the less likely it is that Y is chosen
* The smaller the difference ∆E, the less likely it is that Y is chosen.
* Note the ‘Energy’ is the distance of the Individual tour.
* If the current tour’s distance was greater than the best tour found so far, replace best with it.
* Cool according to some cooling schedule. Described below.
* Back to beginning of loop.

I had an **Individual** class where each individual was a tour of the cities. Contained important methods: generateRandomIndividual() and getDistance() which got the length of the tour.

* ArrayList<Integer> cities = new ArrayList<Integer>(Individual.getTour()

^ How I stored an Individual tour

For best optimisation, when initialising the temperature variable I selected a temperature that initially allowed for practically any move against the current solution. This gives the algorithm the ability to better explore the entire search space before cooling and settling in a more focused region.

**Data Structures and data representations**

* ArrayList<integer> to represent a tour
* Once again only parameter of the run method was the 2D array matrix.

**Experimentation**

The cooling schedule of simulated annealing consists of four components:

-Starting temperature

-Final temperature

-Cooling rate/Temperature Decrement

-Probability of choosing a worse successor

There is a large tradeoff between effectiveness and speed here, if we set the temperate to 100, it would find a solution very quickly but this is very unlikely to be an optimal solution.

I started by altering the probability of choosing a worst successor to low probabilities and high probabilities but in general this made little difference.

**Cooling Schedules**

1. I started with a linear cooling scheme: Tk = Tk – cooling rate.

I started with temperate of 100 and cooling rate of 1.0001.

This gave quick solutions to give me an upper bound to estimate a good but often far from optimal solution.

As I have no limitation on time, for the tours of fewer nodes, I set the temperate to very high (around 100000000) to achieve as close to optimal solution as possible.

I found T0 = 150000000 gave me results for all city files in times under a few minutes.

1. I then swapped to a geometric cooling scheme: T= aT. Where a is a constant less than but close to 1.

From my experience, I found it works well if a is somewhere between 0.8 and 0.99 with better results being found in the higher end of the range. Of course, the higher the value of α, the longer it will take to decrement the temperature to the stopping criterion.

In comparison to schedule 1, I found this to give slightly better results.

I started with doing only 1 iteration at each temperature and decrease the temperature very slowly.

An alternative is to dynamically change the number of iterations as the algorithm progresses. At lower temperatures it is important that a large number of iterations are done so that the local optimum can be fully explored. At higher temperatures, the number of iterations can be less.

i.e. swapping more than 2 elements in the tour for low temperatures.

I altered my algorithm to allow this to achieve better results.

1. I swapped to a logarithmic cooling scheme Tk = (a T0) / ln(1+k)

|  |  |  |
| --- | --- | --- |
| Number Of Cities | Tour A Distance  Greedy Best First Search | Tour B Distance  Simulated Annealing |
| 12 | 56 | 56 |
| 17 | 1636 | 1782 |
| 21 | 2985 | 3263 |
| 26 | 1799 | 1805 |
| 42 | 1573 | 1895 |
| 48 | 14879 | 17831 |
| 58 | 27302 | 38122 |
| 175 | 22311 | 43850 |
| 180 | 5420 | 8930 |
| 535 | 50058 | 51038 |

Where Tk is the value of the temperature at iteration k, To is the initial temperature and α is the cooling speed parameter. I found this cooling schedule to be very slow so I didn’t use it for the larger tours and if I decreased **a** to make it faster it gave worse tours.

**Final Best Results for each implementation**