Simulated Annealing - Introduction:

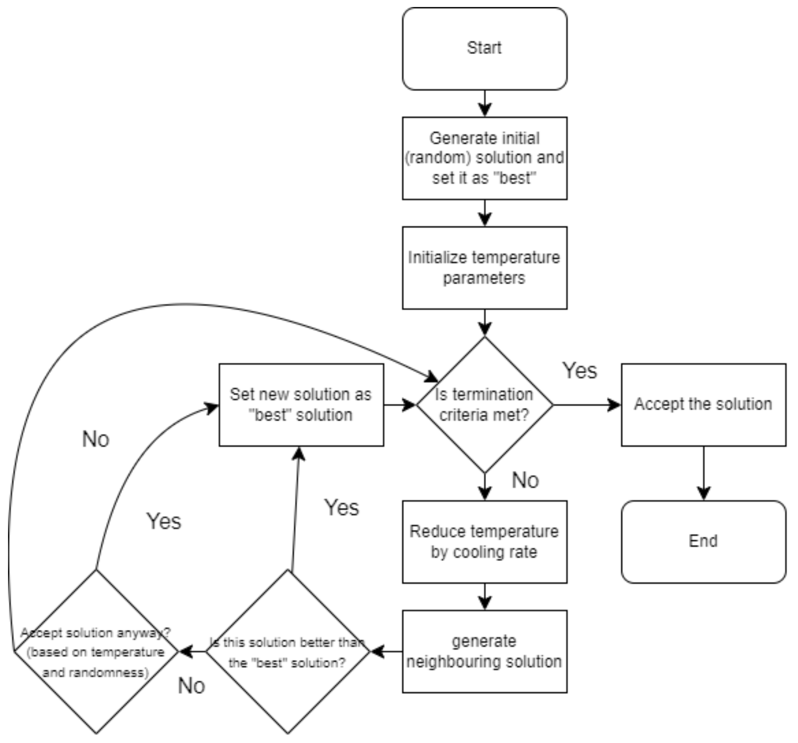
Simulated Annealing, a stochastic global search algorithm inspired by metallurgy annealing (slowly cooling a heated material whilst altering its properties), involves iteratively exploring a neighbouring solution (to the current "best" solution) and progressing with the one with the lowest cost (for example with the TSP, cost is distance). However, some random chance, in addition to a temperature value that gradually decreases as the process continues, may allow a worse solution to be selected, allowing for the possibility of escaping local optima.

Pros:

- Capable of leaving local optima and avoiding noise
- Relatively simple to implement with no complex maths involved and minimal parameters
- It is flexible in that it can handle a diverse range of optimization problems

Cons:

- May be slow to converge on global optimum or an acceptable solution
- Parameters such a temperature and cool-down alpha may take time to tune correctly
- The stochastic nature of the algorithm may lead to greatly inconsistent results across different runs even with the same parameters


Start
Generate initial (random) solution and set it as "best"
Initialize temperature parameters
Is termination criteria met?
Yes
Accept the solution
End
No
Reduce temperature by cooling rate
generate neighbouring solution
Is this solution better than the "best" solution?
Yes
Set new solution as "best" solution
No
Accept solution anyway? (based on temperature and randomness)
Yes
No

```
SimulatedAnnealing(problem, initial_solution, initial_temperature,
cooling_schedule)
    current_solution = initial_solution
    temperature = initial_temperature
    while termination criteria  not met do
        temperature = cooling_schedule(temperature)
        next_solution = generate_neighbor(current_solution)
        delta = evaluate(next_solution) - evaluate(current_solution)
        if delta > 0 then
            current_solution = next_solution
        else
            probability = acceptance_probability(delta, temperature)
            if random() < probability then
                current_solution = next_solution
    return current_solution
```

In my specific implementation of simulated annealing for TSP, the termination criteria was simply an "iteration count" that could be set before execution, ensuring the algorithm would terminate after a specific number of iterations. This controls the overall runtime and allows easier direct comparisons to be made between different runs. Neighbouring solutions were generated using the 2-opt method (swapping two edges in the solution) due to it being simple yet efficient in aiding the algorithm in escaping local minima – the neighbour generated might not always be initially better, but may lead to something better later on. Additionally, the temperature schedule decreased by a constant factor of alpha (for example, 0.9) after each iteration, allowing the algorithm to transform from exploration focused into more exploitation focused over time.

Simulated Annealing - Parameters:

My simulated annealing algorithm has two tuneable parameters: starting temperature and alpha (cool-down rate). The parameters were tuned by starting with values suggested by AM_2017083014324828.pdf (scirp.org), and then making adjustments to them each time (keeping within a range of values outlined in the paper, for example cooling factor should be between 0.8 and 0.99). Only one parameter was changed at a time so that each parameters effect could be clearly identified and attributed, and towards the end, each set of parameters was run twice, to ensure one run didn't just get 'lucky' in finding a low-cost solution. A broader range of values were chosen at first, and then more precise changes were made as the process went on. "Iteration count" was also a parameter that could be changed, but it was kept consistent between each run to allow for a fairer comparison. This value was kept reasonably low (1000), compared to the later runs of 10000, so that the trial runs could be evaluated quickly whilst still being given time to run effectively.

Underlined values are baseline values (from the paper) and **bold** values are the ones that variate from the baseline for that specific run

Trial Number	**Initial Temperature**	**Alpha (cool-down)**	**Iterations**	**Distance (nearest whole number)**
1	100	0.9	1000	46263
2	100	**0.8**	1000	46157
3	100	**0.85**	1000	46996
4	100	**0.99**	1000	48613
5	100	**0.95**	1000	44302
6	**1**	0.9	1000	50128
7	**30**	0.9	1000	46305
8	**50**	0.9	1000	47263
9	**80**	0.9	1000	45560
10	**300**	0.9	1000	47266
11	**500**	0.9	1000	43839
12	**1000**	0.9	1000	47087
13	**500**	**0.8**	1000	43849
14	**80**	**0.8**	1000	43834
15	**500**	**0.95**	1000	44205
16	**80**	**0.95**	1000	42457
17	**50**	**0.95**	1000	48029
18	**80**	**0.9**	1000	43260
19	**80**	**0.99**	1000	43413
20	**85**	**0.95**	1000	44310
21	**85**	**0.95**	1000	46977
22	**75**	**0.95**	1000	43239
23	**75**	**0.95**	1000	45189
24	**75**	**0.92**	1000	42201
25	**75**	**0.92**	1000	46992
26	**75**	**0.97**	1000	45652
27	**75**	**0.97**	1000	46831
28	**80**	**0.95**	10000	34702
29	100	0.9	10000	34851

Iteration count improves performance the higher it gets as it allows the algorithm to explore the solution space for longer. Trials with a higher **initial temperature** tend to perform better up until a point (such as 1000) where performance starts to worsen again – the run needs enough time to explore the solution space extensively, but this needs to be focused to exploitation at a certain point. **Alpha (cool down)** needs to be carefully balanced as too low a value will weaken performance (such as 0.8), as will too high a value (0.99) - either the temperature decreases too quickly and exploration is not allowed or the algorithm explores for too long and doesn't get a chance to exploit better solutions.

Simulated Annealing – Average Results:

Run number	Distance
1	34820
2	35747
3	35049
4	35742
5	35464
6	35978
7	34756
8	34013
9	35307
10	34339
11	35772
12	35204
13	35710
14	35074
15	34880
16	34490
17	36328
18	34568
19	35700
20	36590
21	34910
22	35829
23	34968
24	34401
25	35101
26	35718
27	35316
28	34900
29	35310
30	34994

The parameters used for the 30 independent runs were:

Initial temperature: 80

Alpha: 0.95

Iterations: 10000

(each iteration contained 1 fitness evaluation so 10000 iterations = 10000 fitness evaluations)

From the 30 runs, the results were (to the nearest whole number):

Average distance: 35233

Standard deviation: 590

Genetic Algorithm - Introduction:

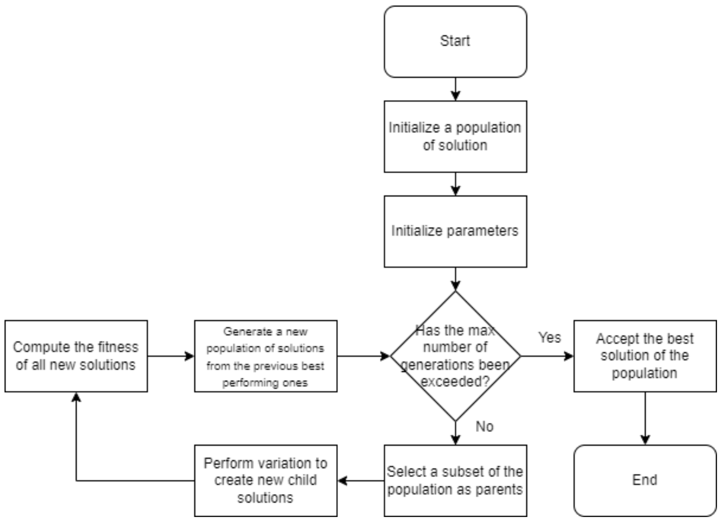
The Genetic Algorithm, inspired by the principles of natural selection (like survival of the fittest), is a heuristic search and optimisation technique. It functions by iteratively evolving a population (group) of possible problem solutions towards achieving more optimal solutions. Evolution is achieved by selecting the best performing solutions of the population, variating them in some way to create new solutions, and then calculating the fitness of all the solution and generating a new population out of the best ones for the process to be repeated.

Pros:

- Searching from a population of solutions instead of a single one increases the chance of finding better solutions even in a large solution space (use of parallel processing)
- Relatively easy to implement whilst being able to handle multiple objectives and constraints
- Works with different types of representation such as binary or permutation

Cons:

- There are even more possible parameters that need to be carefully tuned such as population size, selection method and crossover rates
- Effort needs to be taken to maintain a population diversity so that it is still exploratory
- It may have a high computational cost with large population sizes or complex solutions


Start
Initialize a population of solution
Initialize parameters
Has the max number of generations been exceeded?
Yes
No
Accept the best solution of the population
End
Select a subset of the population as parents
Perform variation to create new child solutions
Compute the fitness of all new solutions
Generate a new population of solutions from the previous best performing ones

```
function GeneticAlgorithm(population_size, num_generations,
crossover_rate, mutation_rate):
    population = initialize_population(population_size)
    for generation = 1 to num_generations:
        evaluate_population(population)
        selected_parents = select_parents(population)
        offspring = crossover(selected_parents, crossover_rate)
        mutated_offspring = mutate(offspring, mutation_rate)
        population = replace_population(population, mutated_offspring)
    return best_individual(population)
```

In my implementation of the genetic algorithm, the selection method was Tournament Selection where you sample a random subset of k solutions and choose the best one. This was for its ease of implementation and lack of reliance on an absolute fitness value – it balances exploration and exploitation. It also provides an extra parameter (tournament size) that can be tuned and explored for the purpose of this assignment. Selected solutions would have a chance of undergoing 1-point crossover (combining two parents) and then inversion mutation (inverse order between two points) for variation. These methods worked well to preserve tour validity whilst working well with the chosen encoding, and inversion mutation works similarly to 2-opt in SA which makes for interesting comparison. Additionally, in order to easily maintain a consistent population size, a new generation consisted of all of the new solutions, and then solutions from the previous generation would make up the remaining spots (reminiscent of Elitism: a lower generational gap). This also ensures that the population never gets worse overall with a cost of slightly weakened exploration. The Solutions were represented as a permutation of city order, encoded directly, meaning each city ID in the solution directly corresponds to a gene in the chromosome that could be swapped and changed. This encoding was intuitive to understand and simplified the operation process.

Genetic Algorithm - Parameters:

My genetic algorithm has five tuneable parameters: population size (number of solutions in each generation), crossover rate (percentage of solutions that get chosen for crossover), mutation rate (chance a solution gets mutated), max generations (how many generations the solution runs for) and tournament size (number of solutions put against each other during selection). The parameters were tuned using a similar method to the tuning of the simulated annealing parameters, this time using https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758 as an initial guideline for values. Because the number of fitness evaluations is determined by population size and max generations, a rule was placed that the two need to multiply to 5000 (and then max generations would be multiplied by 2 to get to 10000 for the later stage runs)

Trial Number	Population Size	Crossover Rate	Mutation Rate	Max Generations	Tournament Size	Distance (nearest whole number)
1	25	0.8	0.2	200	2	45464
2	25	**0.5**	0.2	200	2	59819
3	25	**0.75**	0.2	200	2	55174
4	25	**0.99**	0.2	200	2	47933
5	25	**0.9**	0.2	200	2	46518
6	25	0.8	**0.1**	200	2	61378
7	25	0.8	**0.9**	200	2	48556
8	25	0.8	**0.75**	200	2	45907
9	25	0.8	**0.5**	200	2	46001
10	25	0.8	**0.3**	200	2	48540
11	25	0.8	0.2	200	**3**	50254
13	25	0.8	0.2	200	**5**	52715
14	25	0.8	0.2	200	**6**	49105
15	25	0.8	0.2	200	**7**	48860
16	**8**	0.8	0.2	**625**	2	50498
17	**10**	0.8	0.2	**500**	2	47093
18	**20**	0.8	0.2	**250**	2	45131
19	**50**	0.8	0.2	**100**	2	52976
20	**40**	0.8	0.2	**125**	2	56498
21	**20**	**0.9**	0.2	**250**	2	47307
22	**20**	**0.99**	0.2	**250**	2	47031
23	**20**	0.8	**0.9**	**250**	2	40347
24	**20**	0.8	**0.9**	**250**	2	44159
25	**20**	0.8	**0.75**	**250**	2	38825
26	**20**	0.8	**0.75**	**250**	2	39224
27	**20**	0.8	**0.7**	**250**	2	40251
28	**20**	0.8	**0.7**	**250**	2	40761
29	**20**	**0.9**	**0.75**	**250**	2	40083
30	**20**	**0.9**	**0.75**	**250**	2	41219
31	**20**	**0.85**	**0.75**	**250**	2	38605
32	**20**	**0.85**	**0.75**	**250**	2	40775
33	25	**0.8**	**0.75**	200	2	41976
34	25	**0.8**	**0.75**	200	2	40570
35	**20**	0.8	**0.75**	**250**	**7**	37962
36	**20**	0.8	**0.75**	**250**	**7**	37168
27	**20**	0.8	**0.75**	**250**	**6**	37655
38	**20**	0.8	**0.75**	**250**	**6**	36664
39	**20**	0.8	**0.75**	**250**	**5**	37256
40	**20**	0.8	**0.75**	**250**	**5**	38489
41	**20**	0.8	**0.75**	**500(250*2)**	**6**	34979
42	25	0.8	0.2	400(200*2)	2	38922

The parameter that caused the biggest fluctuation in distance was the **crossover rate** as it also determined how many parents got selected. A higher crossover rate seemed to decrease distance, unless it got too high (like at 0.99). Similarly, a high **mutation rate** helped the algorithm to explore new solutions and prevent premature convergence (leading to lower distances), however a very high mutation rate may have disrupted too many good solutions and weakened the performance. **Tournament size** didn't have a uniform trend, with 2 being the best performing value and then 7 being the second best. This may be because with this implementation, solutions could be selected as parent's multiple times and thus low size encouraged exploration, and high size encouraged exploitation (better solutions would be selected significantly more – multiple times). With regards to the ratio between **population size and max generations**, it performed best when both values were maximised, rather than one being very high and the other being low. For instance, 25:200 performed better than 8:625 or 50:100. You want enough of a population to mix for crossover, as well as enough time (generations) to explore the search space.

Genetic Algorithm - Average Results:

Run number	Distance
1	35833
2	35612
3	36107
4	36380
5	35483
6	35071
7	36091
8	34816
9	34259
10	36704
11	35207
12	33975
13	34898
14	35548
15	34952
16	36924
17	35325
18	35366
19	36028
20	34511
21	37012
22	35876
23	35081
24	35839
25	36013
26	35529
27	35606
28	34218
29	37441
30	36661

The parameters used for the 30 independent runs were:

Population size: 20

Crossover rate: 0.8

Mutation rate: 0.75

Max generations: 500

Tournament size: 6

(number of fitness evaluations depend on population size * maximum generations as each solution undergoes an evaluation every generation)

From the 30 runs, the results were (to the nearest whole number):

Average distance: 35612

Standard deviation: 837

Comparing Simulated Annealing and the Genetic Algorithm:

H0 (null hypothesis): There is no significant difference in the performance (average distance travelled) between my Simulated Annealing Algorithm and Genetic Algorithm

H1 (alternative hypothesis): There is a significant difference in the performance (average distance travelled) between my Simulated Annealing Algorithm and Genetic Algorithm

Run Number	SA Distance	GA Distance	Difference	Absolute Difference	Ranks	Signed Ranks
1	34820	35833	-1013	1013	19	-19
2	35747	35612	135	135	5	5
3	35049	36107	-1058	1058	21	-21
4	35742	36380	-638	638	11	-11
5	35464	35483	-19	19	1	-1
6	35978	35071	907	907	16	16
7	34756	36091	-1335	1335	23	-23
8	34013	34816	-803	803	14	-14
9	35307	34259	1048	1048	20	20
10	34339	36704	-2365	2365	29	-29
11	35772	35207	565	565	10	10
12	35204	33975	1229	1229	22	22
13	35710	34898	812	812	15	15
14	35074	35548	-474	474	9	-9
15	34880	34952	-72	72	3	-3
16	34490	36924	-2434	2434	30	-30
17	36328	35325	1003	1003	18	18
18	34568	35366	-798	798	13	-13
19	35700	36028	-328	328	8	-8
20	36590	34511	2079	2079	26	26
21	34910	37012	-2102	2102	27	-27
22	35829	35876	-47	47	2	-2
23	34968	35081	-113	113	4	-4
24	34401	35839	-1438	1438	24	-24
25	35101	36013	-912	912	17	-17
26	35718	35529	189	189	6	6
27	35316	35606	-290	290	7	-7
28	34900	34218	682	682	12	12
29	35310	37441	-2131	2131	28	-28
30	34994	36661	-1667	1667	25	-25

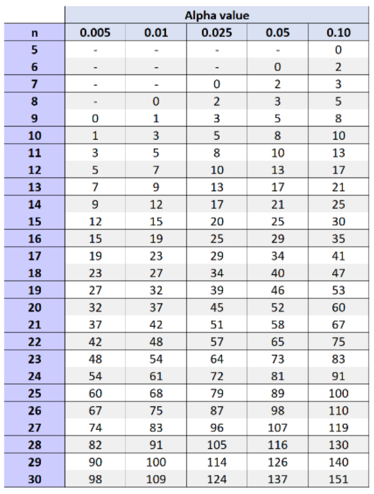
Sum of positive ranks: 150

Sum of negative ranks: 315

W: 150 (min of 150 and 315)

The critical value for W at N = 30 (p < 0.05, two-tailed test) is 137.

150>137 therefor we fail to reject the null hypothesis. We do not have sufficient statistical evidence to say that one algorithm performs significantly better over the other



	Alpha value				
n	**0.005**	**0.01**	**0.025**	**0.05**	**0.10**
5	-	-	-	-	0
6	-	-	-	0	2
7	-	-	0	2	3
8	-	0	2	3	5
9	0	1	3	5	8
10	1	3	5	8	10
11	3	5	8	10	13
12	5	7	10	13	17
13	7	9	13	17	21
14	9	12	17	21	25
15	12	15	20	25	30
16	15	19	25	29	35
17	19	23	29	34	41
18	23	27	34	40	47
19	27	32	39	46	53
20	32	37	45	52	60
21	37	42	51	58	67
22	42	48	57	65	75
23	48	54	64	73	83
24	54	61	72	81	91
25	60	68	79	89	100
26	67	75	87	98	110
27	74	83	96	107	119
28	82	91	105	116	130
29	90	100	114	126	140
30	98	109	124	137	151

Reflection:

While the statistical analysis here may indicate no significant difference between the performance of SA and GA for solving the TSP48, there are other factors to be considered. Despite the similar performance, SA was easier to implement, requiring fewer coding steps and less parameters to tune. Additionally, the parameters of SA were less sensitive and so tuning did not have to be so precise. On the other hand, however, GA may be a lot more scalable when applied to other problem domains due to its ability to explore a larger search space efficiently using population-based evolution. Decisions made may have also significantly impacted performances. For instance, with this GA, a form of Elitism (old solutions in the new generation) was used where maybe using entirely new solutions each generation would have been better instead. Or perhaps a constraint where solutions can only be tournament selected once would have made the performance differ significantly. Overall, this shows that quality should not be the solely judged on performance, and when selecting which algorithm to use, how long you have to implement it or how large/complex the problem is should also be considered, as well as exactly how they are implemented.
Note that an alpha value of 0.05 was selected (see the table above) as it is the most common choice for a task like this. It indicates that you're willing to accept a 5% chance of making a Type I error (incorrectly rejecting the null hypothesis). If a value of 0.1 had been selected then 150<151, and we could reject the null hypothesis and instead accept the alternative hypothesis, however this would have a greater risk of being a false positive, and so was ultimately not selected.