Genetic Algorithm

Overview

For preliminary research and key terminology regarding genetic algorithms see the '*Research*' section.

Every genetic algorithm is comprised of the same few distinct functions. Although, depending on the algorithm's specific purpose, functions in the same theoretical category can be unrecognizable from one another. The fundamental genetic algorithm steps are as follows:

- Initialization: The algorithm starts by creating an initial population of candidate solutions to the problem, referred to as chromosomes.

- Fitness Evaluation: Each chromosome in the population is evaluated based on its fitness, which is a measure of how well it solves the problem.

- Selection: The fittest chromosomes are selected to become the parents of the next generation. This is done using a selection method, such as tournament selection or roulette wheel selection.

- Crossover: The selected individuals are then crossed to create new offspring. This involves exchanging genetic information between the parents to create a new chromosome.

- Mutation: To introduce diversity into the population, some individuals undergo a random mutation. This involves changing a small part of the individual's genetic information.

- Repeat: Steps 2-5 are repeated for a number of generations until a termination condition is met.

Through these steps, genetic algorithms mimic the process of natural selection, where the fittest individuals are more likely to survive and reproduce, passing on their genes to the next generation. This results in, on average, higher fitness (quality) solutions appearing each generation—just as in nature.

Seeding

While implementing my genetic algorithm, I considered whether seeding it with Christofides Algorithm would lead to any efficiency improvements. I theorized that, for larger instances, it would initially slow the algorithm down, but that it would ultimately overcome the lost time by causing the genetic algorithm to converge upon the optimal solution faster. Curious, I looked into the literature but couldn't find any conclusive answers. Unsure of whether to include it in the final solution or not, I opted to conduct my own research.

Pure vs Hybrid

Using *testing.py* (see '*Technical Solution'* section), I compared the performance of the genetic algorithm with and without seeding. I chose the test instances to have twenty nodes to ensure if any improvement had been made, it would be apparent.

Analysis

- The seeded genetic algorithm terminated after 43.12% less time when tested on the same set of coordinates as the pure genetic algorithm.
- The seeded genetic algorithm converged, on average, with a 58.23% lower terminating generation when tested on the same set of coordinates as the pure genetic algorithm.

This data implies that, at least for larger graphs, the seeded genetic algorithm does in fact perform better.

Fitness evaluation

Fitness is typically defied quite generally as a measure of quality for a chromosome.

The quality of a possible travelling salesman solution is fairly simple—primarily dictated by its total edge weight. However, traditionally, greater fitness indicates a better chromosome, and if I were to simply assign total edge weight as fitness, the inverse would be true. To combat this issue, I took the reciprocal of each chromosome's total edge weight as its fitness.

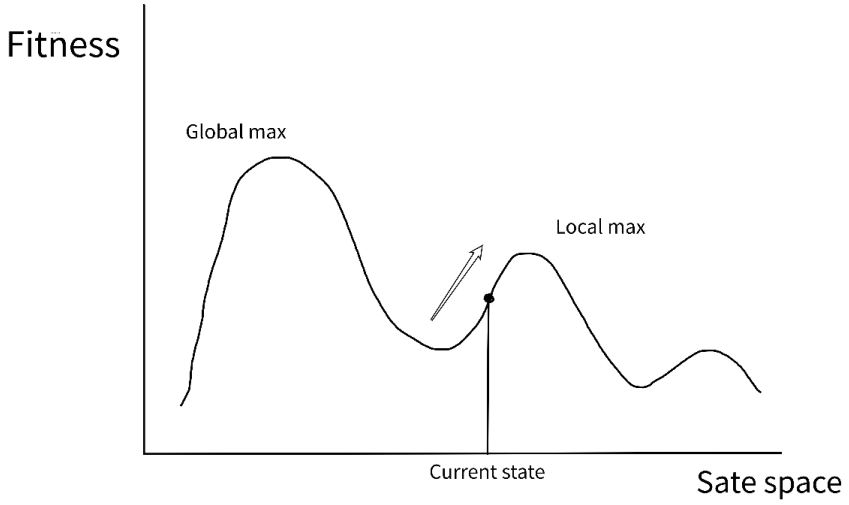
I expect the code that will calculate fitness to be fairly simple and hence not require prototyping—except for the method to find Euclidean distance between two coordinates. The complexity here mostly lies in factoring in the curvature of the earth. Below is the pseudocode I wrote to achieve this task.

```
#Finds distance between two coordinates accounting for the
curvature of the earth

FUNCTION edge_length(c1, c2) THEN #Coordinates are given as a
dictionary

     #Extracts lat and lon values from dictionaries
     lat1, lon1 ← VALUES(c1)
     lat2, lon2 ← VALUES(c2)
     p = pi/180 #Constant to covert between deg and rad

     #Standard formula
     a ← 0.5 - cos((lat2-lat1)*p)/2 + cos(lat1*p) * cos(lat2*p) * (1-
cos((lon2-lon1)*p))/2

     RETURN 12742 * sin(SQUARE_ROOT(a)) #Returns as a float
```

Hill climbing

A genetic algorithm is, by definition, a hill climbing problem. As previously established, a genetic algorithm tries to climb up the fitness landscape by creating a population of potential solutions and then iteratively improving them through selection, mutation, and crossover operations. In each iteration, my algorithm evaluates the fitness of each individual in the population and selects the best ones to pass on their genetic information to the next generation. This selection process biases the algorithm towards better solutions, moving the population up the fitness landscape. This creates a predisposition to tend towards maxima, a trait on which all genetic algorithms function. However, without preventative measures, there is the potential for the algorithm to get stuck at a local maximum, unable to make a big enough change in a single generation to reach the slope of a 'hill' closer to the global maximum.

Below I have drawn a visual illustration of the problem:



Selection

The main responsibility of a selection algorithm is to balance maintaining international randomness with adequately weighting for fitness. To best achieve this compromise, I elected to implement tournament selection. Tournament selection is effective in preventing premature convergence and can handle problems with rugged fitness landscapes.

In this method, a subset of individuals is randomly chosen from the population, and the individual with the highest fitness value among them is selected as a parent. Adjusting the parameter dictating the size of the tournament, i.e., the number of individuals competing, is essential in balancing selection pressure, and diversity in the population. The larger the tournament size the more likely it is the fittest chromosome is included and wins automatically, increasing the weight of fitness in the chance of selection.

I chose to select the second parent based solely on the pseudorandom functions provided by the python library, *random*. I made this decision to ensure adequate intergenerational randomness.

Deciding upon the ideal selection method and then refining it so that the algorithm will actually generate a meaningful result represents a large part of the difficulty in this section of the algorithm. Once it is out of the way, the actual implementation can be fairly straightforward, as demonstrated by the below pseudocode.

```
#picking two chromosomes to cross (tournament selection)
FUNCTION selection(population) THEN

       fitness_weighting ← 0.1
       smple ← sample(population, ROUND(LENGTH(population)*fitness_weighting))

       #x is fittest chromosome in sample
      #y is a random chromosome from the population
       RETURN smple[0][1], RANDOM_CHOICE(population)[1]
```

Crossover

The primary responsibility of a crossover function is to produce the largest fitness improvement from an input of two parent chromosomes. The function can be inconsistent and produce large improvements occasionally or be consistent in generating minor improvements—a good crossover function must be assessed holistically.

I chose to use order crossover. In this technique, a random segment of genes from one parent is selected and copied into the corresponding positions of the other parent while preserving the order of the remaining genes. The resulting offspring will contain the selected segment from one parent and the remaining genes from the other parent, ensuring that the ordering of the genes is maintained, a trait vital for the context of the travelling salesman solution.

```
FUNCTION cross(x, y) THEN #order crossover

       #getting a random subsection of the y chromosome

              last ←  LENGTH(y)-1 #max subsection length
       IF last > 1 THEN
              min ←  2 #min subsection length
       ELSE THEN
              min ←  1
       ENDIF

       sub_len ← RANDOM_INTEGER(min, last) #length of the subsection
       start_i ← RANDOM_INTEGER (0, last-sub_len) #starting index of the subsection

       #allows for 'looping' around the path
       IF (start_i > LENGTH(y)-sub_len) THEN #if subsection exceeds end
              sub ← y[start_i:] + y[:sub_len-len(sub)]
       ELSE THEN
              sub ← y[start_i:start_i+sub_len]
       ENDIF

       #creating a partly empty chromosome with only the subsection in the same place it was
       child ← [None]*start_i+sub+[None]*(last-start_i-sub_len+1)

       #filling the empty spaces in the child chromosome with the remaining nodes in x
       LOOP FOR EACH node IN MULTIPLE_REMOVE(sub, x)
              child[child.index(None)] ← node
        ENDFOR

       RETURN child #returns new chromosome without fitness
```

Mutation

Mutation is the process of randomly changing the values of genes in a chromosome. It is a crucial operator in genetic algorithms that introduces diversity in the population of candidate solutions. It, among other measures, combats the hill climbing nature of the problem by enabling the algorithm to explore new regions of the solution space that may not have previously been discovered.

The effectiveness of mutation in genetic algorithms depends on the parameter of mutation rate, which determines the probability of a gene being mutated. If the mutation rate is too low, the population may converge prematurely. On the other hand, if the mutation rate is too high, the population may lose its good solutions too quickly, leading to poor performance. Finding the ideal mutation rate for my implementation took much trial and error. After much deliberation, I chose to use a scramble mutation, where a subset of genes is shuffled randomly within a chromosome. For a technical breakdown see '*Technical Solution*' section

Controlled variation sources

Although the majority of chromosomes are generated through reproduction, some chromosomes are created not emulating the process of evolution. But, serve to guide the process towards the intended result. These methods account for the differences between the goal of the natural process and the artificial.

Retention

It would be inadvisable to throw away the fittest chromosome between generations in the hope that it is regenerated, or that a better chromosome emerges. For this reason, many genetic algorithms will preserve, at the very least, the fittest chromosome across generations.

In my implementation, I will define a parameter called *maintain_rate* that controls the quantity of the fittest chromosomes are retained across generations. But instead of keeping it the same regardless of the population size, I want it to adapt in response to population size.

A larger value of the *maintain_rate* parameter results in reduced chance of losing the best solution between generations. But equally, it slows the solution down, incentivizing converging upon the first hill encountered. For each additional chromosome maintained across generations, there are decreasing marginal returns on risk, but the resultant reduction in population diversity remains constant. For this reason, as population size increases, maintain_rate must reach a maximum. A consideration accounted for the final implementation.

Random chromosomes

In a final effort to overcome the hill climbing problem, I plan to introduce completely new random chromosomes between generations to keep the algorithm exploring the fitness terrain. I plan to use the same function to create the initial population as to introduce new random chromosomes. With less overall lines of code, I will further my goal of a robust solution.

Termination conditions

The genetic algorithm must have some way of knowing when to stop, this task is achieved by predefined criterion referred to as terminating conditions. In my algorithm I plan to use two different types, as described below.

Maximum generation

I will include an absolute generational limit, dependent on the quantity genes contained within each chromosome. This will ensure that even if the optimal solution isn't found, the algorithm will terminate within a reasonable amount of time.

No improvement

I will include a maximum quantity of generations that can occur without a fitness improvement before the algorithm is terminated. This condition primarily checks whether the optimal solution has been found, at which point no further fitness improvements will be possible.

Implementation

I plan to implement the above conditions as detailed by the below pseudocode.

```
no_improvement ← 0
generation ← 0

GENERATE_INITIAL_POPULATION()

WHILE generation < MAX_GENERATIONS AND no_improvement <
GENERATION_THRESHOLD

          generation ← generation + 1
          CREATE_NEW_GENERATION()

          IF FITTEST(population) == FITTEST(new_generation)THEN
               no_improvement ← no_improvement + 1
           ELSE THEN
                no_improvement ← 0

          population = new_generation

ENDWHILE

return population
```

Parameters

The values of these parameters (all constants) are essential in ensuring that the genetic algorithm work efficiently and effectively.

Parameter	Variable Name	Formula	Description
Population size	k	len(coords)*50	Defines the quantity of chromosomes included in each generation.
Retention	maintain_rate	round(k/100) if k/100>3 else 3	Dictates how many of the fittest chromosomes are preserved across generations.
Generational randomness	generational_randomness	1	Determines how many completely random chromosomes are introduced each generation.
Maximum generations	max_gens	round(k/2)	Maximum generation that can be reached before the algorithm is automatically terminated.
Generation threshold	generation_threshold	round(max_gens/5)	Maximum generations without improvement
Mutation rate	mutation_rate	0.2	Dictates the probability that any given individual child chromosome will mutate.
Fitness weighting	fitness_weighting	0.1	Determines the size of the tournaments in the selection processes as a percentage of the population.

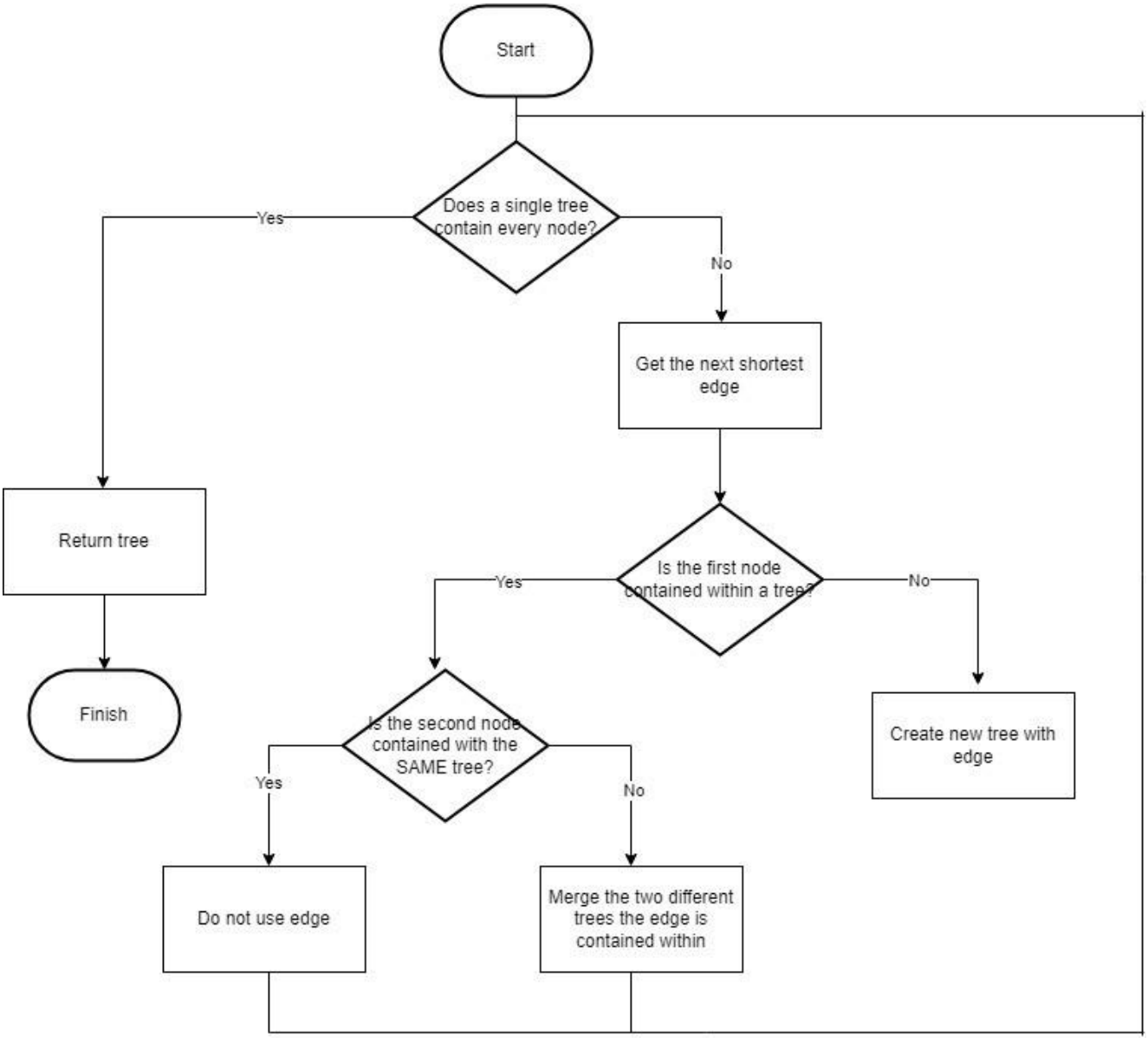
Christofides Algorithm

Christofides algorithm can be dissected into few distinct sections, as done below.

Kruskal's Algorithm

The first step of Christofides algorithm is to find the minimum-spanning-tree of the given graph. Although any algorithm can theoretically be used, Kruskal's algorithm is often utilized as the efficient go-to solution.

Below is a flowchart representing Kruskal's algorithm. When I created the flowchart I omitted the initialization of a forest of trees at the beginning, mistakenly thinking it was assumed.

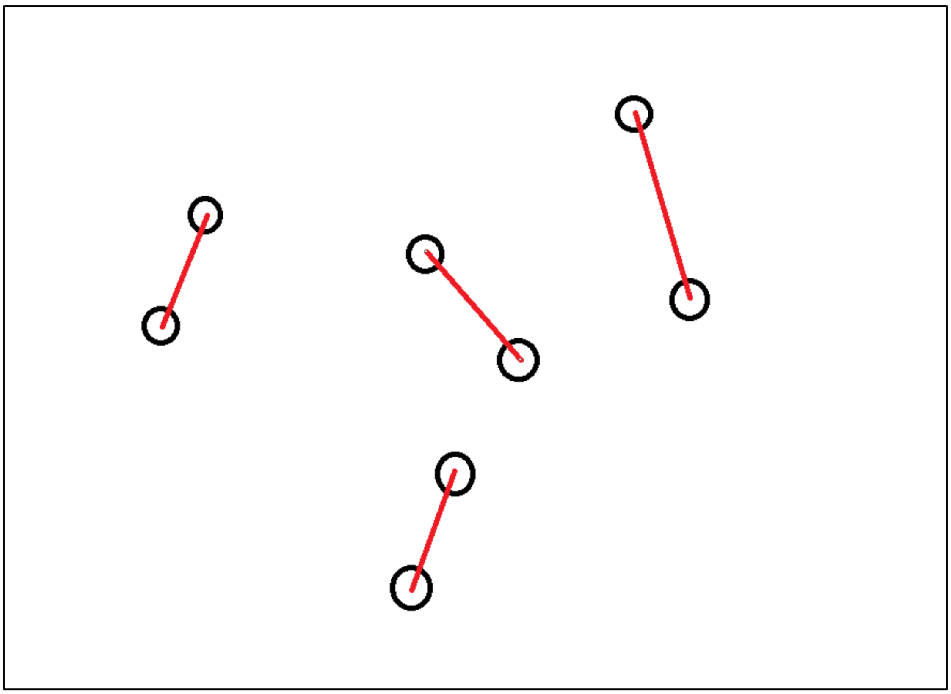

Start
Does a single tree contain every node?
Yes
No
Return tree
Finish
Get the next shortest edge
Is the first node contained within a tree?
Yes
No
Is the second node contained with the SAME tree?
Yes
No
Do not use edge
Merge the two different trees the edge is contained within
Create new tree with edge

Minimum Perfect Matching

The next step is to find the minimum perfect matching solution for the same graph. I think the best way to explain what minimum perfect matching it to break it down to the sum of its parts:

Minimum	Refers to the fact that the total weight or cost of the edges in the set is minimized. This means that there is no other matching of edges that covers all the vertices with a lower total weight or cost.
Perfect	Refers to the fact that each vertex in the graph is matched with exactly one edge from the set. This means that the matching is complete and there are no unmatched vertices.
Matching	Refers to a set of edges that cover all the vertices of the graph such that no two edges share a common endpoint. In other words, each edge in the matching is incident with a different pair of vertices.

For example, the minimum perfect matching of a graph may look something like the following I have drawn:

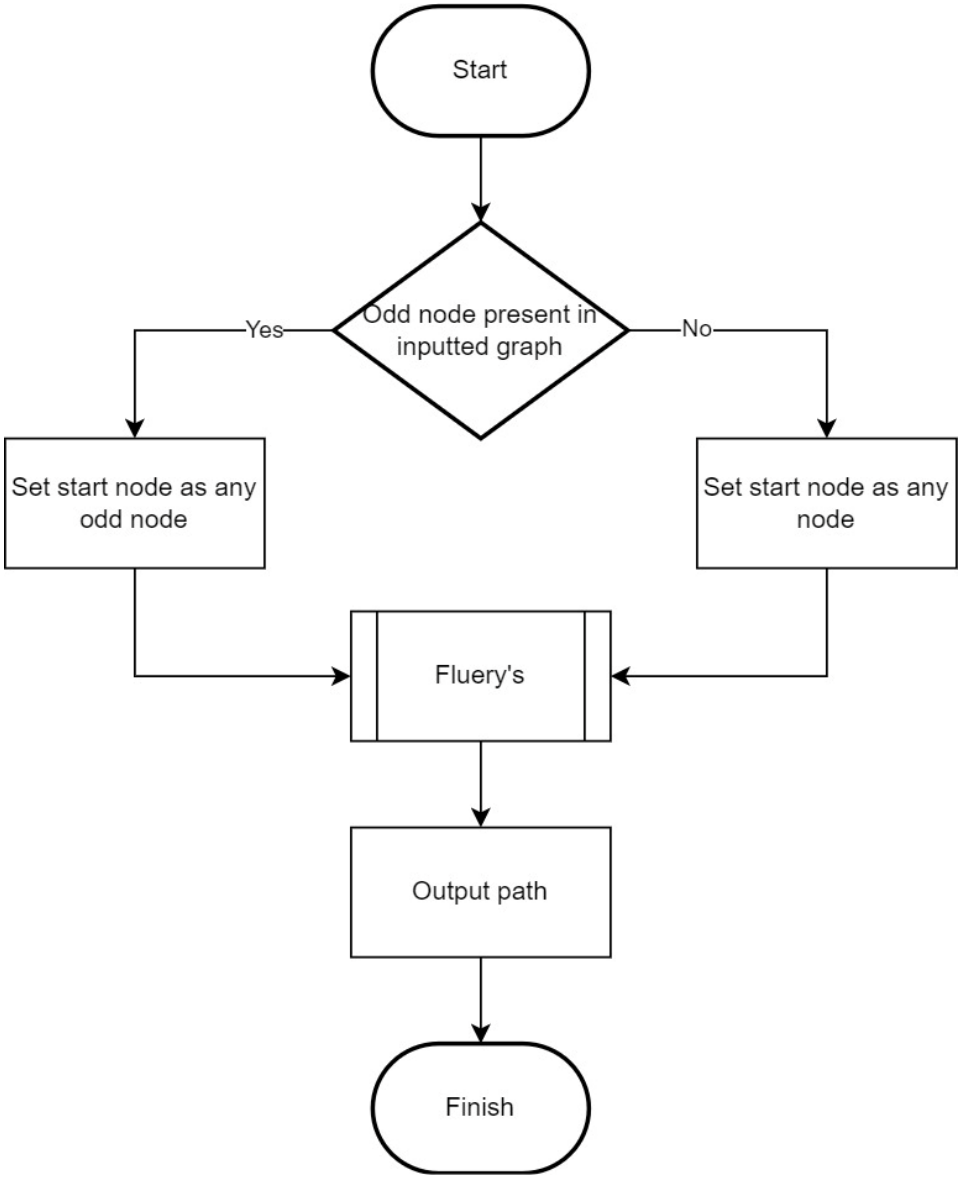


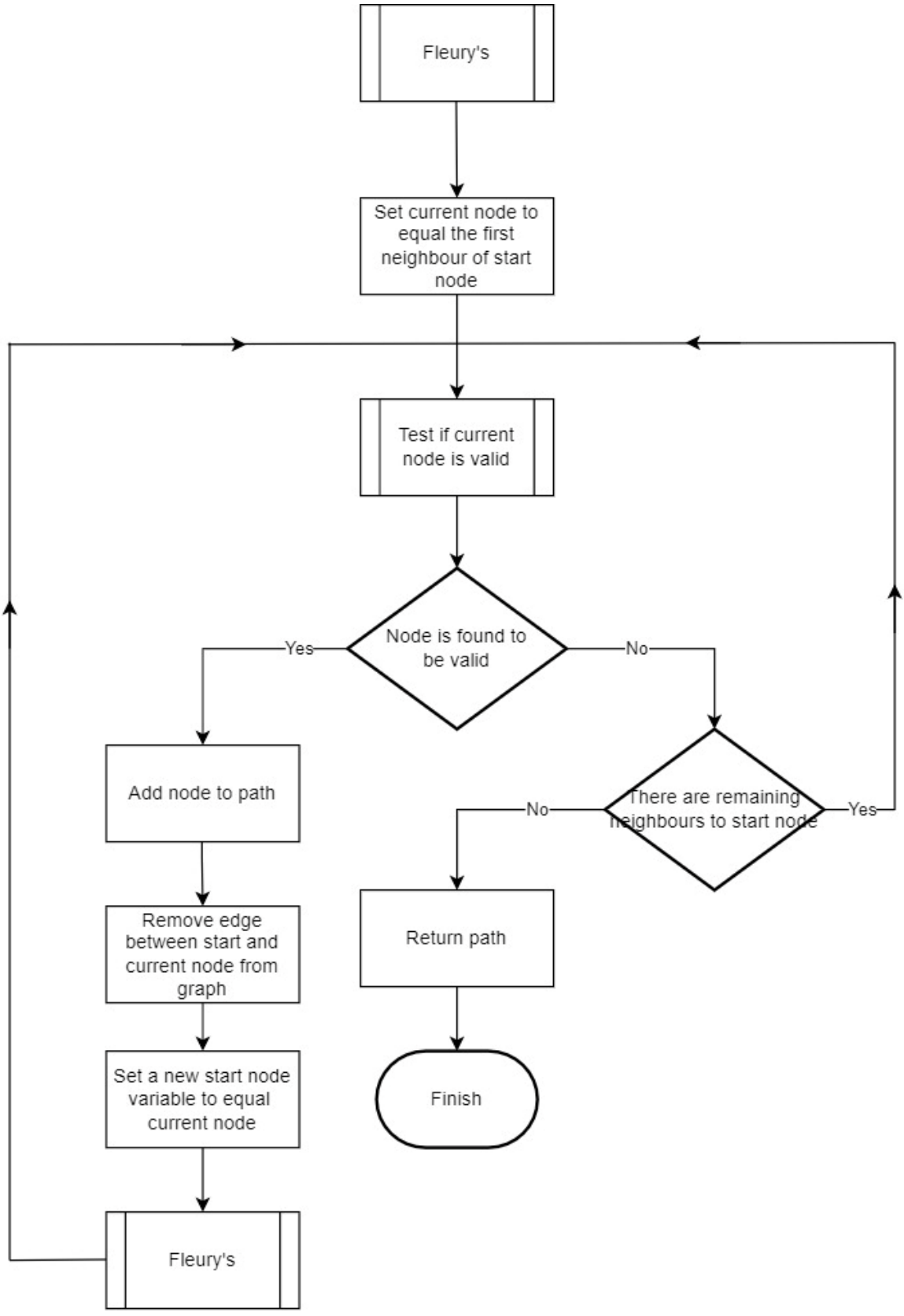
Fleury's Algorithm

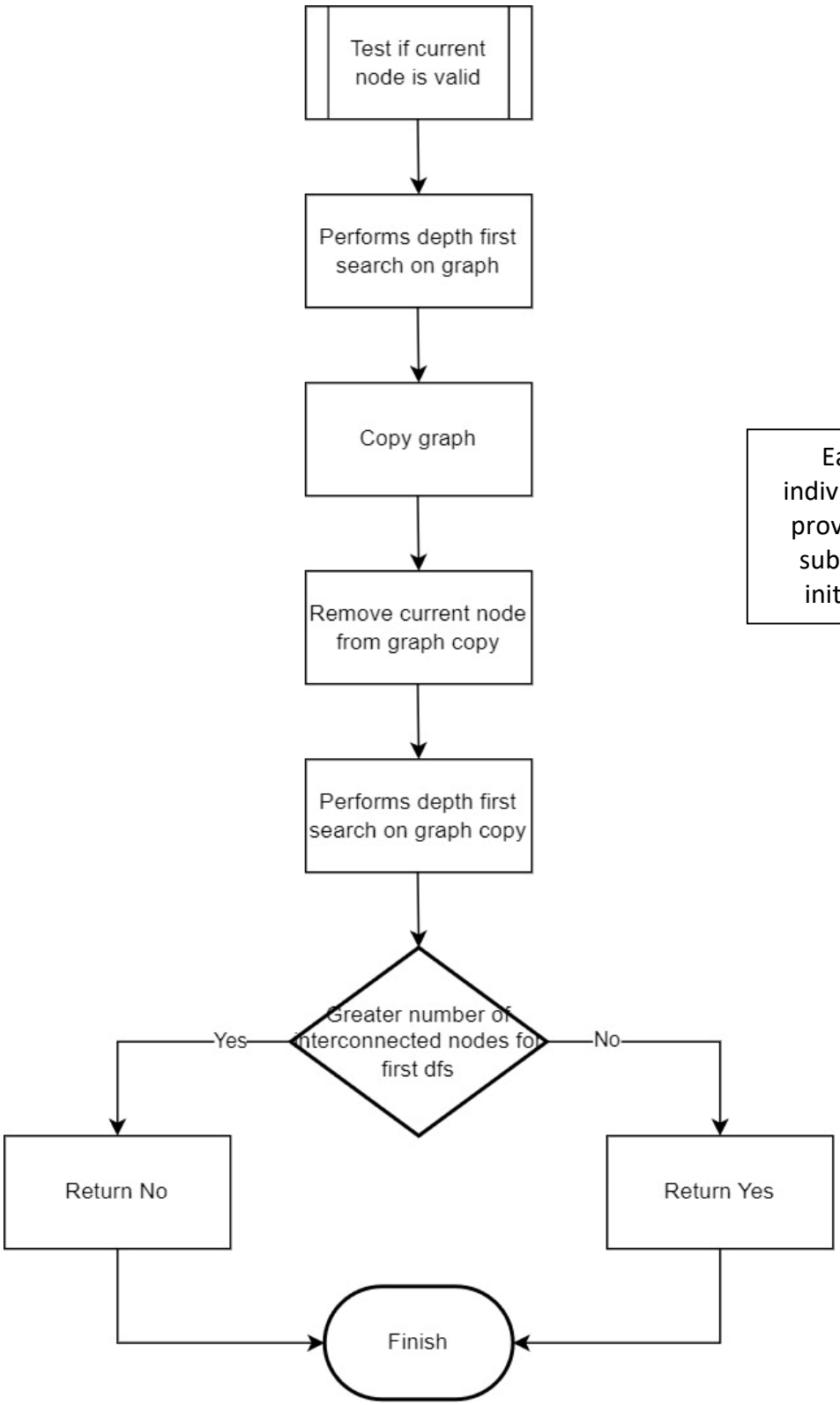
While nearing the finish line of implementing Christofides algorithm, I ran into an unexpected hurdle – one unmentioned during my preliminary research.

Much of the algorithm deals in graph processes and translations, but once these have been completed one is left with an adjacency matrix, not a path. Realizing this discrepancy, I researched how to convert between what I had, a graph, and what I needed, the solution to the travelling salesmen problem. I then realized that what I was, in fact, after was the graph's Euler circuit. Using the new piece of terminology as a search term, I was met with Fleury's algorithm, which although considered broadly outdated, I then implemented as a means of a solution.

The below flowchart shows how I approached the problem of implementation.




Fleury's
Set current node to equal the first neighbour of start node
Test if current node is valid
Node is found to be valid
Yes
No
Add node to path
There are remaining neighbours to start node
No
Yes
Remove edge between start and current node from graph
Return path
Set a new start node variable to equal current node
Finish
Fleury's



Each of these individual processes proved to be more substantial than I initially thought.