Ant Colony Optimization

Jaraad Kamal

1 Introduction

The Traveling Salesman Problem (TSP) is a classic problem in computer science and mathematics. The task is to determine the optimal route/tour through a series of towns such that a salesman can visit all of them while minimizing the total distance traveled. The problem was first formally studied in the 1930s by Karl Menger. Historical records show that it was first mentioned in 1832 [Yu14]. This problem has major applications (one being postage and package delivery routes).

Traveling Salesman Problem is believed that determining an exact solution to the problem cannot be completed in polynomial time. All current methods of finding exact solutions to the problem require on the order of $\mathcal{O}(n!)$ computations where n is the number of towns [Yu14]. This makes it is infeasible to find an exact solution even for small values of n such as 20.

There are some methods to get an approximate solution very quickly. One such approach is Ant Colony Optimization. This approach developed by Dorigo and Gambardella in 1997 simulates a group of ants to determine a sufficient solution to TSP [DG97]. Sebastian Lague provides a beautiful visualization of this approach [Lag21].

2 Definitions

Discussion of the algorithm requires the following definitions. A *graph* is a series of points (or towns) in a space. A *vertex* is a particular point (or town) on this graph. An *edge* is a connection (or road) between two vertices on a graph. A *cost* is the price (or distance) associated with a particular edge. Finally a *path* (or tour) is a sequence of edges that visits different vertices on a graph without visiting the same one twice.

3 Implementation

3.1 Outline

The basic approach for the algorithm is to simulate a number of ants that are scattered randomly along the graph and will randomly tour the graph.

The decisions the ants makes are probabilistic. It will randomly choose an edge to follow but will prefer edges with lower cost. In this sense the tour is "semi-random" while also being a greedy algorithm because it prefers lowest cost at every step.

At the end of the first round of tours the ants will deposit a *pheromone* along the path it took in its individual tour. The amount of *pheromone* deposited depends on total cost of the ants path. Ants with lower total costs will deposit more pheromones along each edge they crossed.

Then during the next round new ants will then semi-randomly tour the graph. However they will both prefer cheaper edges as well as paths that have pheromones deposited on them. This helps reinforce "better" paths.

At the end of each round a portion of the pheromones will get *evaporated* (i.e. a the amount of pheromones on each path will be reduced). This evaporation is to prevent paths from becoming permanent. Additionally, evaporation helps bad paths to fade out after multiple rounds.

3.2 Notation

There are a lot of variables and parameters that can be adjusted when implementing the algorithm. For

the sake of brevity I will define these as follows:

- m - The number of ants
- a - The pheromone desirability weight, $a \geq 0$
- b - The path desirability weight, $b \geq 1$
- p - The pheromone evaporation factor, $0 < p < 1$
- q - The pheromone deposition factor, $q > 0$
- η_{xy} - The desirability of edge (x, y). Typically inverse weight (e.g. $1/dist(x, y)$).
- τ_{xy} - The amount of pheromone on edge (x, y)

3.3 Formulas

3.3.1 Edge Selection

Let U be the set of all vertices that have not been visited by the ant.

The probability an ant will travel from vertex x to vertex y (given $y \in U$):

$$P_{xy} = \frac{(\tau_{xy})^a (\eta_{xy})^b}{\sum_{z \in U} (\tau_{xz})^a (\eta_{xz})^b} \tag{1}$$

[22]

3.3.2 Pheromone Update

Let l_k be the total cost of the tour of ant k. I will define $\Delta\tau_{xy,k}$ as:

$$\Delta\tau_{xy,k} = \begin{cases} q/l_k & \text{If ant } k \text{ traveled on edge } (x, y) \\ 0 & \text{Otherwise} \end{cases} \tag{2}$$

When updating the pheromones we can account for evaporation and new deposition at the same time with:

$$\tau_{xy} \leftarrow (1 - p)\tau_{xy} + \sum_{k}^{m} \Delta\tau_{xy,k} \tag{3}$$

[22]

3.4 Thoughts and Modifications

The one major difference between the algorithm and my implementation is in the determination of edge probabilities. During the first few rounds of the algorithm the pheromone strength (τ_{xy}) on most of the edges is zero. Thus numerator of equation (1) is zero and there is no probability to explore a new edge. In my implementation I changed the probability equation by adding a constant to τ_{xy} to ensure a non-zero probability:

$$P_{xy} = \frac{(1 + (\tau_{xy})^a)(\eta_{xy})^b}{\sum_{z \in U} (1 + (\tau_{xz})^a)(\eta_{xz})^b} \tag{4}$$

After testing I found that the most significant factors are the pheromone evaporation factor, pheromone desirability weight, and path desirability weight. The number of ants did not cause big changes. Results after using 5 ants and 20 ants were comparable. Additionally there are diminishing returns on the number of rounds/iterations. Running 500 iterations and 1000 iterations caused little differences.

My default values are:

m	5 - 10
a	2
b	5
p	0.2
q	5

Figure 1: Default values chosen for algorithm

3.5 Pseudo Code

The algorithms are split into two major chunks. The first algorithm called Ant Brain is the basic decision tree for each simulated ant. The second algorithm called Ant Optimize will set up the ants each round and keep record/modify the pheromones.

Algorithm 1: Ant Brain

```
Data: graph, pheromones
Function ant_brain(G, PH: matrix, start: int) :
    path ← [];
    visited ← {} ; /* set of visited nodes */
    current ← start;
    cost ← 0;
    for i = 0, ..., (n − 2) do
        weights ← [];
        /* possible next locations */
        choices ← {x|x ∉ visited ∧ x ≠ start};
        /* weights for weighted random choice */
        for j ∈ choices do
            τ_j ← 1/G[current, j];
            η_j ← PH[current, j];
            /* a and b are predefined constants */
            append(weights, (1 + τ_j^a) * η_j^b);
        end
        /* weights should sum to 1 */
        weights ← 1/(Σ w_j) (weights);
        /* getting next node */
        next ← random(choices, weights);
        /* updating current, cost, path, and visited */
        append(path, (current, next));
        cost ← cost + G[current, next];
        visited ← visited ∪ {next};
        current ← next;
    end
    /* adding path back to start */
    append(path, (current, start));
    cost ← cost + G[current, start];
    return(cost, path);
end
```

Algorithm 2: Ant Colony Optimization

```
Data: graph
Function ant_optimize(G: matrix, num_ants: int, num_iter: int) :
    n ← size(G);
    /* normalizing graph */
    G ← (1/max(G))*G;
    /* Setting up pheromone matrix */
    PH ← zeros((n, n));
    ants ← [];
    for i = 0, ..., (num_iter − 1) do
        for a = 0, ..., (num_ants − 1) do
            st ← randint(0, n - 1);
            append(ants, ant_brain(G, PH, st));
        end
        /* Evaporating pheromones with predefined p */
        PH ← (1 - p) * PH;
        /* Depositing new pheromones with predefined q */
        for ant ∈ ants do
            path ← ant.path;
            cost ← ant.cost;
            for step ∈ path do
                (s, e) ← step;
                /* PH must be symmetric */
                PH[s, e] ← PH[s, e] + q/cost;
                PH[e, s] ← PH[e, s] + q/cost;
            end
        end
    end
    best_cost ← min(ants).cost;
    best_path ← min(ants).path;
    return(best_cost, best_path);
end
```

4 Results

In order to test the algorithm I created a random map of 52 coordinates (Figure 2). Fifty-two coordinates were chosen because it is large enough to be infeasible to obtain a an exact solution to TSP.

I then ran the algorithm with 5 ants and 10 itera-

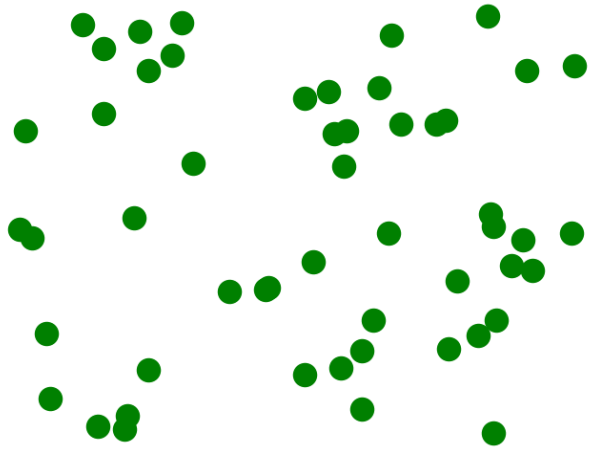


Figure 2: Test map of points for Ant Colony Optimization. The average distance between two points is 102.20 units.

tions (Figure 3) as well as 1000 iterations (Figure 4) to see if there was any difference. After 10 iterations the solution had a total cost of 1595.23 units while the 1000 iteration solution had a cost of 1311.08 units. To visualize the ant decisions I plotted the pheromone trails left by the ants (Figure 5).

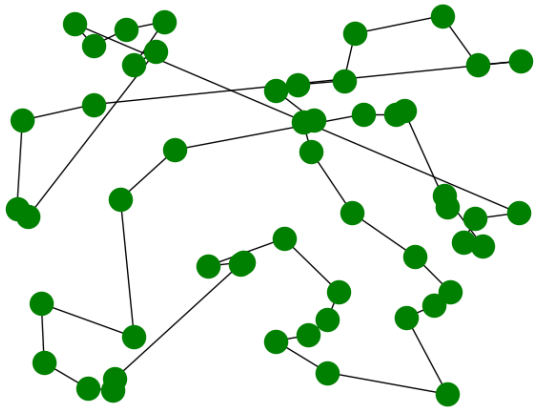


Figure 3: Solution produced by 5 ants over 10 iterations. Total cost is 1595.23 units. Time spent searching: 0.1 seconds.

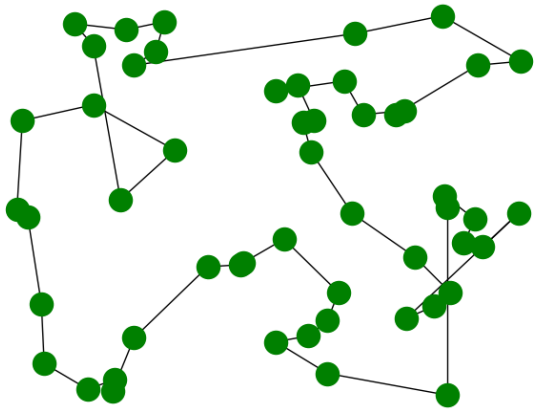


Figure 4: Solution produced by 5 ants over 10 iterations. Total cost is 1311.08 units. Time spent searching: 8.6 seconds.

Figure 5: Pheromone trails after 5 ants and 1000 iterations. The darker lines represent edges that are more saturated with pheromones and thus more preferable to ants.

The figures show that the algorithm is finding a visually optimal path across all the vertices. There are big cycles to avoid unnecessary movement. Additionally, the second iteration (which had 100x the rounds) did find a better solution. However upon

close inspection the increased iterations only reduced the solution cost by 300 units. this is an indication of diminishing returns.

Overall the algorithm is very fast. As previously mentioned the TSP on a 52 vertex graph is computationally infeasible. This approach (with 1000 iterations) it only took 8.6 seconds to compute.

5 Discussion

The issue with this algorithm is that is probabilistic. Different runs of the algorithm can produce slightly different results. There will also always be a probability that the given output is inefficient. However the greedy nature of each ant attempts to avoid this issue.

Another issue is that there is no way to verify that a given output is THE solution to the traveling salesman problem on the graph. The only method is to compute the exact solution which is impossible for anything other than a tiny graph.

The absolute biggest benefit to this approach is that it is very fast. For larger graphs this is incredibly fast than a true brute force approach. One of the only current methods for finding a true solution to the graph is calculating each possible. This approach takes around $\mathcal{O}(n!)$ computations where n is the number of vertices [Yu14]. A graph of 52 vertices (as in the example) would require $\mathcal{O}(52!)$ computations. To put the 52! into perspective, it is more than the number of seconds since the beginning of the universe [06]. Even for a graph with 10 solutions an exact answer is infeasible to compute.

There are similar insect based algorithms. One such algorithm is called Bee Colony Optimization. This procedure attempts to simulate a swarm of bees in order to solve multi-dimensional optimization problems [Kar05]. While it does not solve the TSP like Ant Colony Optimization it uses a similar approach.

The benefit to Ant Colony Optimization is that approaches TSP in polynomial time. In the example a solution to a 52 vertex graph was found in 8.6 seconds. An infeasible problem was given an approximate solution in a matter of seconds.

References

[06] *Imagine the universe!* 2006. URL: https://imagine.gsfc.nasa.gov/science/featured_science/tenyear/age.html.

[22] *Ant colony optimization algorithms*. 2022. URL: https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms.

[DG97] M. Dorigo and L.M. Gambardella. "Ant colony system: a cooperative learning approach to the traveling salesman problem". In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 53–66. DOI: 10.1109/4235.585892.

[Kar05] Dervis Karaboga. "AN IDEA BASED ON HONEY BEE SWARM FOR NUMERICAL OPTIMIZATION". In: 2005.

[Lag21] Sebastian Lague. *Coding adventure: Ant and slime simulations*. Mar. 2021. URL: https://www.youtube.com/watch?v=X-iSQQgOd1A.

[Yu14] Jessica Yu. *Traveling Salesman Problems*. 2014. URL: https://optimization.mccormick.northwestern.edu/index.php/Traveling_salesman_problems.

Ant Colony Optimization

Jaraad Kamal

In [ ]:
```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import random
from threading import Thread
```

The Ant

Creating a structure to hold information needed by each ant

In [ ]:
```python
class AntInfo:
    def __init__(
            self,
            start_pos: int,
            graph: np.matrix,
            graph_dist_desir: np.matrix,
            pheromone_trails: np.matrix,
            pheromone_desirability_weight: int,
            indices: list[int],
            ones: np.matrix
            ):

        # indicies is a shared list from 0 to max_indices that is used by all
        # ants. SHOULD BE UNTOUCHED
        self.INDICES = indices

        # ones is a shared list 1s used by
        # ants. SHOULD BE UNTOUCHED. Idea was because do not want to creat
        # a new matrix every time an ant is created (reduced speed)
        self.ONES = ones

        # phermone desirability weight is to account for how much each ant
        # prefers pheremones (SAME FOR ENTIRE CLASS)
        self.pheromone_desirability_weight = pheromone_desirability_weight

        verts, verts2 = graph.shape
        if verts != verts2:
            raise ValueError('matrix must be square')
        if graph_dist_desir.shape != (verts, verts2):
            raise ValueError('Graphs must be same size')
        if (verts, verts2) != pheromone_trails.shape:
            raise ValueError('pheromones must be same shape')

        self.num_vertices = verts
        self.start_pos = start_pos
        self.graph = graph
        self.graph_dist_desir = graph_dist_desir
        self.pheromone_trails = pheromone_trails

        # value that will hold the path found by the thread
        self.return_val = None
```

The Ant Logic

The following is the basic ant brain. This code represents the decision tree that each individual ant will undergo. From a given starting point it will randomly choose an edge. It will perfer edges that are closer (shorter distances) and have high amounts of pheromones

In [ ]:

```
def ant_follow(ant_info: AntInfo) -> None:
    """
    Given an ant info parameter it will start at a position and
    pseduo-randomly follow paths to complete the graph.

    When selecting next edge to follow it will chose one that goes to an
    unvisited node. Then it will do a weighted probability based on the
    current pheromone strength and the distance to the point (perfering
    stronger pheromone and shorter distances)

    The calculations will be stored in the ant_info parameter under return
    val. This is to get information out of the threaded application.

    THREAD SAFE
    """
    # keeping track of visited locations
    visited = [1] * ant_info.num_vertices
    # we visit first by default
    visited[ant_info.start_pos] = 0
    current_pos = ant_info.start_pos
    path = [None] * (ant_info.num_vertices)
    total_cost = 0
    pheromone_bias = lambda x: np.power(x, ant_info.pheromone_desirability_weight)

    for i in range(0, ant_info.num_vertices - 1):
        #
        # desirabilities is the desiability from the given graph * pheromones
        # which is done element by element



        # each individual desirability should be:
        # (inverse_dist ^ des_weight) * (pheromone_trail_val ^ pheromone_weight)
        # the exponents should be already accounted in the matrices.
        # the adding 1 to all pheromone values is to prevent pheromone values
        # from ever hitting zero and then having the entire weight be 0
        desirabilities = np.multiply(
            ant_info.graph_dist_desir[current_pos, :],
            np.add(
                ant_info.ONES,
                pheromone_bias(ant_info.pheromone_trails[current_pos, :])
            )
        )
        # multiplying by visited because if visited is 0 then we have been there
        # and the probability goes to 0
        desirabilities = np.multiply(desirabilities, visited)

        # weird tolist()[0] is because random.choices only accepts lists
        # and because desiribility is a matrix the tolist() function will
        # return a list of lists instead of just a single row list
        next_node = random.choices(
            ant_info.INDICES, desirabilities.tolist()[0])[0]
        # updating visited
        visited[next_node] = 0
        path[i] = (current_pos, next_node)
        total_cost += ant_info.graph[path[i]]
```

```
            current_pos = next_node

        # now we must return back to start
        path[-1] = path[-2][1], ant_info.start_pos
        total_cost += ant_info.graph[path[-1]]

        # putting the return val in the ant_info parameter
        ant_info.return_val = total_cost, path
```

The Algorithm

This will set off a bunch of ants to find an approximate solution to the traveling saleman problem. See comments for details.

Will return a float and 2 matrices. - The float is the total cost by of the journey for the best ant. - The first matrix is the best path found at the end of all iteration. - The second matrix is the pheromone matrix that was the final phermone trail

In [ ]:

```
def ant_optimization(
        graph: np.matrix,
        num_ants: int = 10,
        num_iterations: int = 100,
        desirability_weight: float = 2.0,
        pheromone_desirability_weight: float = 1,
        pheromone_deposition_factor: float = 5.0,
        pheromone_evaporation_factor: float = 0.8,
        thread: bool = False) -> (float, np.matrix, np.matrix):

    """
    Function that sets up ant colony optimization.
    It will set up a series of ants determined by num_ants to explore the
    graph using the ant_folow algorithm.

    Each of the ants will start on a random
    vertex. The amount each ant prefers the closer node is determined by the
    desirability_weight. Ants will also perfer paths that are soaked in
    pheromones (pheromones are left after an ant in a previous iteraion followed
    the edge in question). The amount they prefer
    pheromones is determined by pheromone_desirability_weight.

    It will then evaporate the pheromones on all paths. This is to prevent
    too much reinforcement to potentially bad starting guesses. The evaporation
    rate is determined by the pheromone_evaporation_factor

    Then after it is all finished it will go through each of the paths
    the ants took and deposite pheromones in a separate matrix to indicate to
    future ants that the path is good. The amount of pheromones desposited
    is determined by pheromone_deposition factor.

    NOTE:
        graph > 2x2 symmetrix matrix positive on everything but diagonal
        num_ants > 0
        num_iteraions > 0
        desirability_weight > 1
        pheromone_desirability_weight > 0
        pheromone_deposition_factor > 0
        pheromone_evaporation_factor: positive < 1
        thread -> usually slower if small number of ants

    NOTE:
        overflow errors are due to either the pheromone not evaporating enough
        (0.8 is a good factor) or the pheromone_desirability_factor being too
        high.


    """

    num_verts, _num_verts = graph.shape
    if num_verts != _num_verts:
        raise ValueError('Graph must be a square graph')

    # making sure matrix is symmetric. (meaning graph is undirected)
    if not np.all(graph == graph.T):
        raise ValueError('Graph must be undirected')
```

```
# making sure all entry values are less than 1
max_val = graph.max()
graph = (1 / max_val) * graph

# putting a 1 in the diagonal just caust it makes checking for non zero
# distances really fast
graph = graph + np.identity(num_verts)

# checking if all non zero
if not np.all(graph):
    raise ValueError('Graph must be completely connected')

# doing the subtraction once to get the percentage of pheromones that stay
# after each round
evaporation_factor = 1 - pheromone_evaporation_factor

# we will first convert all the distances in graph into raw desirablily
# values. This will basically take the inverse distance and raise it
# to the desirability weight
dist_to_desire = lambda x: np.power((1/x), desirability_weight)

# necessary info for running the ants.
pheromones = np.zeros((num_verts, num_verts))
graph_dist_desir = dist_to_desire(graph)
INDICES = [i for i in range(0, num_verts)]
ONES = np.ones((1,num_verts))

# setting up each thread
threads = [None] * num_ants
ant_infos = [None] * num_ants
ant_ranks = [None] * num_ants

for iteration in range(0, num_iterations):
    for i in range(0, num_ants):
        ant_info = AntInfo(
            start_pos=random.randint(0, num_verts - 1),
            graph=graph,
            graph_dist_desir=graph_dist_desir,
            pheromone_trails=pheromones,
            indices=INDICES,
            pheromone_desirability_weight = pheromone_desirability_weight,
            ones=ONES
        )
        ant_infos[i] = ant_info

        if thread:
            threads[i] = Thread(target=ant_follow, args=[ant_info])
            # starting the thread
            threads[i].start()
        else:
            ant_follow(ant_info)

    if thread:
        # stopping the threads
        for t in threads:
```

```
            t.join()

        # we just ran all the ants for one round.
        # we must now update the pheromone trails.

        # first we need to add some 'evaporation' to the pheromones on the
        # current trail so that paths don't get permanetly reinforced.
        pheromones = evaporation_factor * pheromones

        for index, ant in enumerate(ant_infos):
            # getting the path cost and path from the ant
            ant_cost, path = ant.return_val
            pheromone_additon = pheromone_deposition_factor / ant_cost

            # ranking the ants only if on last iteration
            if iteration == num_iterations - 1:
                ant_ranks[index] = ant_cost
            for (row, col) in path:
                # row should never equal column because we dont travel to
                # same node from node
                pheromones[col, row] += pheromone_additon
                pheromones[row, col] += pheromone_additon


    # getting the best from the last run
    best_ant = np.argmin(ant_ranks)
    best_price, path = ant_infos[best_ant].return_val
    best_path = np.zeros((num_verts, num_verts))
    for (row, col) in path:
        best_path[col, row] = 1
        best_path[row, col] = 1

    return best_price * max_val, best_path, pheromones
```

Demonstration

In [ ]:

```
def draw_graph(
        matrix: np.matrix,
        pos: dict=None,
        show: bool= True,
        weighted: bool= True,
        file_name: str=None,
        color: str='b',
        color_map=plt.cm.Blues) -> dict:
    """
    Given a matrix it will draw a graph with the weights. It will return the
    positions dictionary so that multiple graphs can have same node positions.
    """
    graph = nx.from_numpy_matrix(matrix)
    if pos is None:
        pos = nx.spring_layout(graph)
    if weighted:
        edges , weights = zip(*nx.get_edge_attributes(graph,'weight').items())
        nx.draw(graph, pos, node_color=color,
            edgelist=edges, edge_color=weights, edge_cmap=color_map)
    else:
        nx.draw(graph, pos, node_color=color, edge_cmap=color_map)

    if file_name is not None:
        plt.savefig(file_name)
    if show:
        plt.show()
    return pos
```

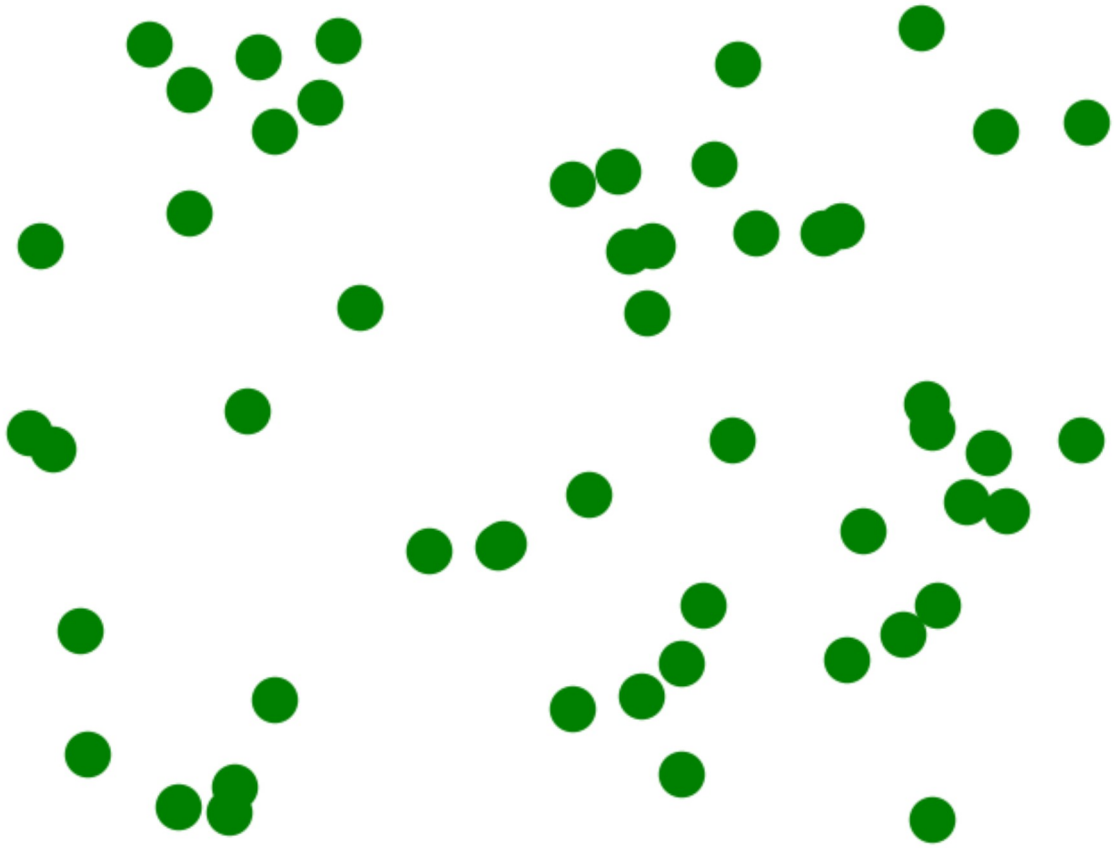
In [ ]:
```
def create_rand_graph(
        size: int,
        max_coord: int=200) -> (np.matrix, list[tuple[int]]):
    """
    Given a size it will return a symmetric matrix representing the graph.
    The matrix is 'random'
    """

    # getting list of positions
    positions = {
        i: (random.randint(0, max_coord), random.randint(0, max_coord))
        for i in range(0, size)
    }

    graph = np.zeros((size, size))

    # calculating distances
    for i in range(0, size):
        for j in range(i, size):
            (x1, y1) = positions[i]
            (x2, y2) = positions[j]
            dist = (x1 - x2) ** 2 + (y1 - y2) ** 2
            dist = np.power(dist, 0.5)
            graph[i,j] = dist
            graph[j,i] = dist

    return graph, positions
```

Creating a Graph

Creating a random graph and visualizing.

In [ ]:
```
# making graph with 52 towns
z, positions = create_rand_graph(52)
```

In [ ]:
```
# passing in zeros because I don't want any edges drawn
_ = draw_graph(np.zeros(z.shape), positions, weighted=False,
    color='g', file_name='Ant_52_nodes_map')
print('average cost per edge =', np.average(z))
```



```
average cost per edge = 102.19582005404104
```

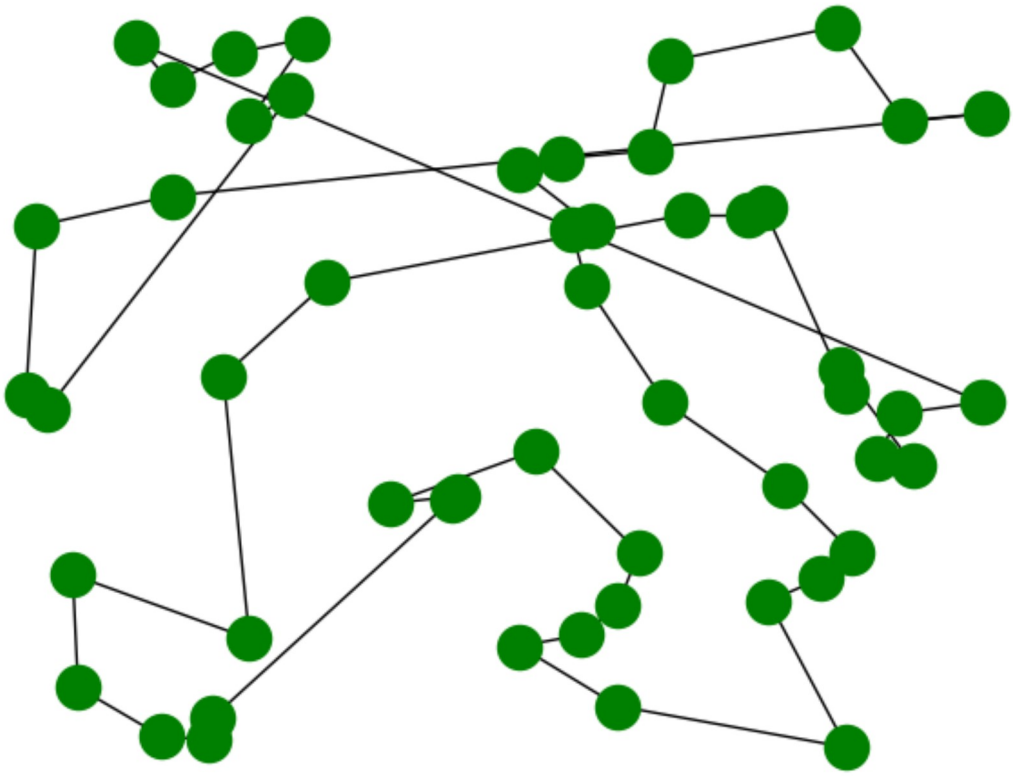
Using Algorithm

10 Iteration Solution

In [ ]:
```
cost, best_path, pheromone_trails = ant_optimization(
    z, num_ants=5,
    num_iterations=10,
    pheromone_desirability_weight=2,
    pheromone_evaporation_factor=0.2
)
print('cost of journey:', cost)
```

```
cost of journey: 1595.2327900432347
```

In [ ]:
```
_ = draw_graph(
    best_path, pos=positions,
    color='g', color_map=plt.cm.Greens,
    weighted=False, file_name='Ant_52_nodes_10_iterations')
```

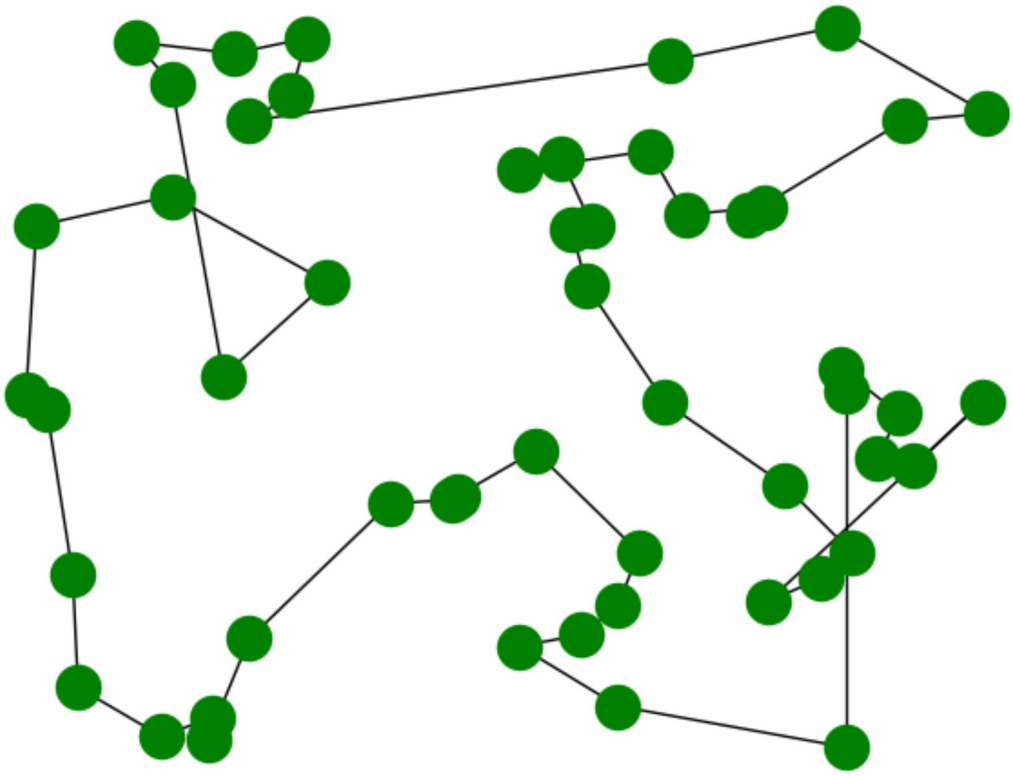


1000 Interation Solution

In [ ]:
```
cost, best_path, pheromone_trails = ant_optimization(
    z, num_ants=5,
    num_iterations=1000,
    pheromone_desirability_weight=2,
    pheromone_evaporation_factor=0.2
)
print('cost of journey:', cost)
```

```
cost of journey: 1311.0816240477477
```

In [ ]:
```
_ = draw_graph(best_path, pos=positions,
    color='g', color_map=plt.cm.Greens,
    weighted=False, file_name='Ant_52_nodes_1000_iterations')
```



Visualizing Pheromone Trails

Out of curisoity lets look at the pheromone trails to get a sense of what each ant is seeing.

The darker an edge the more pheromones on the edge. These are the edges that the ants will prefer as they travel.

In [ ]:
```
_ = draw_graph(pheromone_trails, pos=positions,
    color='g', color_map=plt.cm.Greens,
    file_name='Ant_52_nodes_1000_iterations_pheromones')
```