American College of Thessaloniki

MATH 220

Discrete Mathematics

Submitted To:
Mr. Kostas Karagiannis
Instructor (Mathematics)
Division of Science & Technology

Submitted By:
Paschalis Skouzos
20140340
Spring II 2018

Contents

1 Presentation of the Problem, Collection and Description of Data

1.1 Presentation of the Problem

> **satellite** • a small body that orbits a larger one, particularly the natural satellites of the planets. A natural satellite is also known informally as a moon. [1]

As any other celestial object, physical satellites merit our attention for both idealistic as well as practical reasons. Exploring the cosmos feeds our innate need for knowledge, inspires the next generation of pioneers, and produces technological innovation such as new communication techniques or navigation applications [2]. Apart from our moon, earth's sole physical satellite, Jupiter's and Saturn's systems have also been of particular interest. Numerous flybys have already been performed [3] and a thorough investigation of Jupiter's moon Europa is planned by NASA in order to identify whether the icy moon could harbour conditions suitable for life [4].

1.2 Choosing Destinations

For our project, we will choose six Jovian satellites. We suppose that our space probe's journey from Earth to Jupiter is handled by another department and our job starts once it enters Jupiter's system. Our mission is to perform one flyby of each of those six satellites in order to collect information and exit from the same point from which we entered. Therefore, in order to save time and money we have to optimise our journey. The six satellites are Io, Ganymede, Callisto, Thebe, Themisto, and Himalia.

1.3 Collecting Data

In order to simplify the system we assume that the satellites are stationary and they lie on a circle with radius equal to that of the semi-major axis of their actual, elliptical, orbit [5].

Our space probe is similar to Juno, NASA's probe which was launched in August 2011 and is currently orbiting Jupiter [6]. As a result our orbiting speed will be 3425 km/h, the number was observed on 02/05/2018 [7]. Therefore, we have to make six assumptions:

Name	**Radius** (10^3 km)
Io	421.8
Ganymede	1070.4
Callisto	1882.7
Thebe	221.9
Themisto	7507.0
Himalia	11460.0

Table 1: Selected satellites

1. The position of the satellites is randomly chosen on their respective orbiting circles.
2. The satellites are stationary.
3. Our probe's initial position is random on a circle with radius $13\,000 \times 10^3$ km and Jupiter as its centre.
4. Our probe moves in a straight line.
5. Our probe, Jupiter, and the satellites lie on the same plane.
6. Our probe, Jupiter, and the satellites are single points.

1.4 Project Statement

In summary, we are entering Jupiter's system in a distance of $13\,000 \times 10^3$ km from the planet, we have to flyby Io, Ganymede, Callisto, Thebe, Themisto, and Himalia, finally we have to exit from the same point from which we entered. The above must be organised in a way that minimises our travel's distance in order to avoid any damage to our probe but also to keep the cost of the project as low as possible.

2 Discussion and Presentation of Course Concepts, Translation of the Problem

2.1 Historical Information

The foundations of graph theory were laid by Leonhard Euler in 1736. A river in Königsberg, Prussia, split the city into four parts which were connected by seven bridges as shown in Figure 1. The problem was to create a path in which each bridge would be crossed only once [8]. More than a century later, in 1878, the term "graph" was officially introduced [9] and the first book on the subject, Kőnig's *Theorie der endlichen und unendlichen Graphen*, was published in 1936 [10].

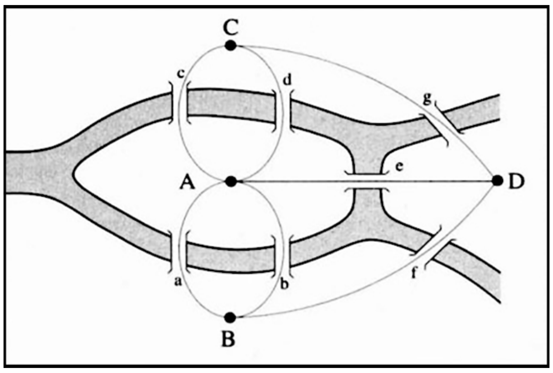

Figure 1: Seven bridges at Königsberg

Since then, graph theory has evolved into an interdisciplinary tool, used in almost every problem which can be expressed as edges and vertices. In computer science, graphs can be used in order to understand the dependency of modules in a large codebase. In aviation, the airlines can model their routes and fleet as a graph in order to find optimal departure times and destinations. In every sport and competition, tournaments usually start as a round-robin where all teams face each other, and the final phase is a single-elimination, both of those models can be illustrated as graphs.

Because of the widespread use of graph theory, research is still performed and many open problems are yet to be solved. A lot of the conjectures deal with graph colouring, a simple conjecture under that category is the following:

> **The 1-2-3 Conjecture** • for every connected graph G of order 3 or more, each edge of G can be assigned one of the colours 1, 2, 3 in such a way that the induced colours of every two adjacent vertices are different.

Although, the conjecture holds for an infinite class of graphs, it is yet to become a theorem. In addition while there is a 1-2-3-4-5 Theorem, even a 1-2-3-4 Theorem is elusive [11].

2.2 Basic Notions and Results

In this section we will present some basic definitions and theorems which will be later used in order to solve our problem.

graph • a *graph* $G = (V, E)$ consists of V, a nonempty set of *vertices* (or *nodes*) and E, a set of *edges*. Each edge has either one or two vertices associated with it, called its *endpoints*. An edge is said to *connect* its endpoints.

degree of a vertex • the *degree of a vertex in an undirected graph* is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by $\deg(v)$.

complete graphs • a *complete graph on n vertices*, denoted by K_n, is a simple graph that contains exactly one edge between each pair of distinct vertices.

path • informally, a *path* is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of the graph. As the path travels along its edges, it visits the vertices along this path, that is, the endpoints of these edges.

Euler circuit & path • an *Euler circuit* in a graph G is a simple circuit containing every edge of G. An *Euler path* in G is a simple path containing every edge of G.

theorem 1 • a connected multigraph with at least two vertices has an Euler circuit if and only if each of its vertices has even degree.

theorem 2 • a connected multigraph has an Euler path but not an Euler circuit if and only if it has exactly two vertices of odd degree.

Hamilton path & circuit • a simple path in a graph G that passes through every vertex exactly once is called a *Hamilton path*, and a simple circuit in a graph G that passes through every vertex exactly once is called a *Hamilton circuit*. That is, the simple path $x_0, x_1, \ldots, x_{n-1}, x_n$ in the graph $G = (V, E)$ is a Hamilton path if $V = \{x_0, x_1, \ldots, x_{n-1}, x_n\}$ and $x_i \neq x_j$ for $0 \leq i < j \leq n$, the simple circuit $x_0, x_1, \ldots, x_{n-1}, x_n, x_0$ (with $n > 0$) is a Hamilton circuit if $x_0, x_1, \ldots, x_{n-1}, x_n$ is a Hamilton path.

Dirac's theorem • if G is a simple graph with n vertices with $n \geq 3$ such that the degree of every vertex in G is at least $n/2$, then G has a

Hamilton circuit.

Ore's theorem • if G is a simple graph with n vertices with $n \geq 3$ such that deg(u) + deg(v) $\geq n$ for every pair of nonadjacent vertices u and v in G, then G has a Hamilton circuit.

2.3 Translation of the Problem

We consider the position in which we enter Jupiter's system our initial vertex v_1 and every moon is also a vertex which we must visit $\{v_2, \dots, v_7\}$. Therefore, our graph has $n = 7$ and each pair is connected since we can directly travel between them. In addition, each edge has a weight equal to the distance of the vertices. As a result, we have a complete graph $K_7 = (V, E)$ with weighted edges as shown in Figure 2.

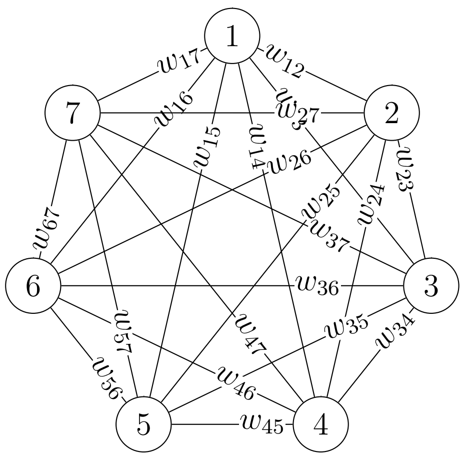

Figure 2: Abstract graph of our problem

Since, $\forall v \in V(deg(v) = 6)$, according to theorem 1 we can conclude that our graph has both an *Euler circuit* and an *Euler path*. Furthermore, using *Dirac's theorem* we can conclude that it has a *Hamilton circuit* because $n = 7$ so $n \geq 3 \wedge \forall v \in V(deg(v) \geq n/2)$. The existance of a Hamilton circuit can also be proven due to *Ore's theorem* since $\forall u \forall v \in V$(deg($u$) + deg($v$) $\geq n$).

Our objective is to find the Hamilton circuit, starting for v_1, with the lowest sum of weights. In graph theory terms we have a travelling salesman problem (TSP).

3 Optimisation

3.1 Optimisation Algorithms

There are two categories of algorithms in order to tackle travelling salesman problems, exact and heuristic. While the former algorithms find the best solution, the latter require less time in order to yield an approximation. Indicatively, some of them are: brute force, branch and bound, simulated annealing, genetic, pairwise exchange, and nearest neighbour.

Nearest neighbour, the one in which we will focus, is a greedy algorithm which performs local optimisation. At each step of our tour we are choosing as our next destination the nearest, unvisited, vertex.

Algorithm 1 Nearest Neighbour for N cities

```
Input: $A$ {$N \times N$ matrix with distances}, $s$ {index of initial vertex}
Output: $C$ {circuit starting from $s$}
 1: $C := [s]$ {initialise our solution $C$}
 2: $current\_vertex := s$
 3: $unvisited\_vertices :=$ array of all N vertices
 4: remove $s$ from $unvisited\_vertices$
 5: $cost := 0$ {initialise our circuit's cost}
 6: for $i := 1$ to $N - 1$ do
 7:    $next\_vertex := unvisited\_vertices[1]$
 8:    $min := A[current\_vertex][next\_vertex]$
 9:    for all $j$ in $unvisited\_vertices$ do {find nearest vertex}
10:       if $A[current\_vertex][j] < min$ then
11:          $next\_vertex := j$
12:          $min := A[current\_vertex][j]$
13:    $C.append(next\_vertex)$
14:    remove $next\_vertex$ from $unvisited\_vertices$
15:    $current\_vertex := next\_vertex$
16:    $cost := cost + min$
17: $C.append(s)$ {account for our return to $s$}
18: $cost := cost + A[current\_vertex][s]$
19: return $C$
```

In order to calculate the complexity of our algorithm we are counting only the comparison in line 10. From our initial vertex s there are $N - 1$ unvisited vertices so the loop in line 9 runs $N - 1$ times. The next time it runs $N - 2$ times, then $N - 3, N - 4 \ldots$, until it finally runs only once. As a result we have:

$$f(n) = (n-1) + (n-2) + \ldots + 2 + 1 < n + n + \ldots + n + n = n(n-1) < n^2$$

Therefore, for $n > 1$, $f(n) < g(n)$ where $g(n) = n^2$. Consequently our algorithm is $O(n^2)$ with witnesses $k = 1$ and $C = 1$. In line 9 we could omit checking the first index, since we already defined it as the nearest neighbour, but the ending complexity would be the same. From our result, we can conclude that our greedy algorithm is a good approach for finding an initial, sub-optimal, solution.

3.2 Applying the Algorithm

In order to test our greedy algorithm, we are going to randomly initialise the position of our objects i.e. probe and moons, and try to minimise our travel with respect to distance. We will choose a number as a seed for our random number generator so that we can produce the same initial configuration for any algorithm we want to test. The input and output of our program is displayed below, the code is listed in A.2 Nearest Neighbour and some auxiliary functions are in A.1 Utilities.

input.txt

```
name, radius(10^3km)
entry, 13000
Io, 421.8
Ganymede, 1070.4
Callisto, 1882.7
Thebe, 221.9
Themisto, 7507
Himalia, 11460
```

NN_output.txt

```
Nearest Neighbour Results
-------------------------
N = 7

Distances Matrix(10^3 km):
         |    entry|       Io| Ganymede| Callisto|    Thebe| Themisto|  Himalia|
    entry|     0.00| 13311.59| 12150.33| 12229.63| 12780.30| 20488.88| 24160.06|
       Io| 13311.59|     0.00|  1223.10|  2285.65|   587.33|  7179.32| 11261.55|
 Ganymede| 12150.33|  1223.10|     0.00|  2291.56|   880.66|  8344.44| 12482.39|
 Callisto| 12229.63|  2285.65|  2291.56|     0.00|  1818.80|  8677.26| 11931.86|
    Thebe| 12780.30|   587.33|   880.66|  1818.80|     0.00|  7723.37| 11678.59|
 Themisto| 20488.88|  7179.32|  8344.44|  8677.26|  7723.37|     0.00|  5413.33|
  Himalia| 24160.06| 11261.55| 12482.39| 11931.86| 11678.59|  5413.33|     0.00|
Nearest Neighbour Circuit:
entry -> Ganymede -> Thebe -> Io -> Callisto -> Themisto -> Himalia -> entry
Total Distance: 54154.62*10^3km

Performance Results
-------------------
Repetitions: 10000
Total time required (s): 4.92E-1
Average time of one execution (s): 4.92E-5
```

3.3 Finding the Optimal Solution

In order to find the optimal solution we are going to run a brute force algorithm which calculates the total distance of all possible circuits. The input is the same as the one used for our nearest neighbour algorithm. The results are displayed below and the code is in A.3 Brute Force. Again, we used the utilities from A.1 Utilities.

BF_output.txt

```
Brute Force Results
-------------------
N = 7

Distances Matrix(10^3 km):
         |    entry|       Io| Ganymede| Callisto|    Thebe| Themisto|  Himalia|
    entry|     0.00| 13311.59| 12150.33| 12229.63| 12780.30| 20488.88| 24160.06|
       Io| 13311.59|     0.00|  1223.10|  2285.65|   587.33|  7179.32| 11261.55|
 Ganymede| 12150.33|  1223.10|     0.00|  2291.56|   880.66|  8344.44| 12482.39|
 Callisto| 12229.63|  2285.65|  2291.56|     0.00|  1818.80|  8677.26| 11931.86|
    Thebe| 12780.30|   587.33|   880.66|  1818.80|     0.00|  7723.37| 11678.59|
 Themisto| 20488.88|  7179.32|  8344.44|  8677.26|  7723.37|     0.00|  5413.33|
  Himalia| 24160.06| 11261.55| 12482.39| 11931.86| 11678.59|  5413.33|     0.00|

Brute Force Circuit:
entry -> Ganymede -> Thebe -> Io -> Themisto -> Himalia -> Callisto -> entry
Total Distance: 50372.46*10^3km

Performance Results
-------------------
Repetitions: 100
Total time required (s): 8.17E-1
Average time of one execution (s): 8.17E-3
```

As we can observe from our results, the greedy algorithm is faster since one execution requires 4.92×10^{-5} s while the brute force approach executes in 8.17×10^{-3} s. Since we have $N = 7$ and a fixed initial vertex, the number of all possible routes is the number of permutations of the 6 remaining vertices $P(6, 6) = 6!/(6 - 6)! = 720$. Hence, the brute force algorithm has to compare 720 total distances. In general, for N vertices we have to make $(N-1)!$ comparisons which results in $O(n!)$ complexity, far greater than the $O(n^2)$ we found for nearest neighbour in 3.1. However, as it was expected, brute force was able to provide the optimal solution which was about 7% better.

3.4 Evaluating your Work

In order to experimentally verify the efficiency of our computations we tested the two algorithms for N ranging from 2 to 12. The code can be found in A.4 Complexity Experiment and the results verify their theoretical counterparts. The complexity of the brute force method renders it computationally heavy even for $N = 11$ while the sub-optimal approach of the greedy algorithm keeps on producing results in less than a millisecond. As a consequence, for any $N > 15$ we would opt for the nearest neighbour algorithm. Although our vertices were limited in a circle with radius $13\,000 \times 10^3$ km, extending it would not bear any consequences since the radius does not affect the number of comparisons. However, we have to be careful not to exceed the maximum integer Python can handle.

analysis_output.txt

```
Analysis Results
----------------

n: number of vertices
NN: average time(seconds) in order to complete the nearest neighbour algorithm
NN trials: number of times the algorithm was run for n
BF: average time(seconds) in order to complete the brute force algorithm
BF trials: number of times the algorithm was run for n

n  |        NN| NN trials|        BF| BF trials|
2  |   6.43E-6|    100000|   9.09E-6|    100000|
3  |   1.31E-5|    100000|   1.71E-5|    100000|
4  |   1.93E-5|    100000|   5.11E-5|     10000|
5  |   3.70E-5|     10000|   2.02E-4|      1000|
6  |   4.37E-5|     10000|   1.15E-3|      1000|
7  |   5.65E-5|     10000|   7.68E-3|       100|
8  |   7.71E-5|     10000|   6.06E-2|        10|
9  |   8.21E-5|     10000|   5.34E-1|        10|
10 |   1.07E-4|     10000|   5.32E+0|        10|
11 |   1.21E-4|     10000|   5.87E+1|        10|
12 |   1.58E-4|     10000|   6.81E+2|        10|
```

In a real world mission, since space agencies have enormous computational power available, they may be using an optimised version of the brute force method, even for $N > 15$. However, a low limit for N probably still exists. The nature and high cost of those missions renders the greedy approach dangerous because of its probability to produce a circuit far from the optimum. For large N, space agencies are likely using an algorithm outside the scope of our course.

Still, if we had to re-do the project, we could also create a better brute force algorithm by checking half of the circuits since in our permutations we have each circuit in normal and reverse order, both of which have the same cost. In addition, we could utilise the speed of our probe, implement a circular motion for our moons and create a set-up closer to the real world.

3.5 Future Trajectories

Each year Google organises a coding challenge called Hash Code. In 2018, the online qualification round was an optimisation problem where a fleet of F self-driving cars had to perform N rides which were determined by their start and end intersections, earliest start and latest finish. Each successful ride scored extra if it started on time or finished earlier than expected. If modelled properly, we could use the routing algorithms in order to tackle it. However, some extra rules must be implemented in order to account for the time restrictions.

Appendix A Code

A.1 Utilities

utilities.py

```python
import math
import random
import sys
from decimal import Decimal


class Moon:
    def __init__(self, name, radius):
        self.name = name
        self.radius = radius
        self.theta = math.radians(random.randrange(360))
        self.x = self.radius * math.cos(self.theta)
        self.y = self.radius * math.sin(self.theta)

    def cartesian_distance(self, moon):
        delta_x = self.x - moon.x
        delta_y = self.y - moon.y
        distance = math.sqrt(math.pow(delta_x, 2) + math.pow(delta_y, 2))
        return distance

    def __str__(self):
        return self.name

    def __hash__(self):
        return hash(str(self))

    def __eq__(self, other):
        return str(self) == str(other)


def read_input(file):
    #set seed for randomising the theta of vertices
    random.seed(1)

    vertices = []
    with open(file, 'r') as f:
        next(f)  # skip first line
        for line in f:
            name = line.split(",")[0]
            radius = float(line.split(",")[1])
            vertex = Moon(name, radius)
            vertices.append(vertex)
    return vertices


def calc_distances(vertices):
    distances = {}
    for starting_vertex in vertices:
        distances_aux = {}
        for ending_vertex in vertices:
            distances_aux[ending_vertex] =
            ↪  starting_vertex.cartesian_distance(ending_vertex)
        distances[starting_vertex] = distances_aux
    return distances


def print_alg_results(alg_name, distances, circuit, cost, timeit_results):
    titlecased_name = alg_name.title()
    initials_of_name = "".join(word[0].upper() for word in alg_name.split())

    # redirect all prints to our file
    sys.stdout = open(initials_of_name + "_output.txt", "w")

    print(titlecased_name + " Results")
    print("-" * len(titlecased_name + " Results"))
    print("N = " + str(len(circuit) - 1))
    print()
    print("Distances Matrix(10^3 km):")
    # print headers
    print('{:>9s}'.format(""), end="|", flush=True)
    for moon in distances:
        print('{:>9s}'.format(moon.name), end="|", flush=True)
    print()
    # print distances
    for starting_moon in distances:
        print('{:>9s}'.format(starting_moon.name), end="|", flush=True)
        for ending_moon in distances:
            print('{:9.2f}'.format(distances[starting_moon][ending_moon]),
            ↪  end="|", flush=True)
        print()
    print()

    print(titlecased_name + " Circuit:")
    for i in range(len(circuit) - 1):
        print(circuit[i], end=" -> ", flush=True)
    print(circuit[-1])
    print("Total Distance: " + '{:.2f}'.format(cost) + "*10^3km")
    print()
    print("Performance Results")
    print("-" * len("Performance Results"))
    print("Repetitions: " + str(timeit_results[0]))
    print("Total time required (s): " +
    ↪  "{:.2E}".format(Decimal(timeit_results[1])))
    print("Average time of one execution (s): " +
    ↪  "{:.2E}".format(Decimal(timeit_results[1]/timeit_results[0])))
```

A.2 Nearest Neighbour

nearestneighbour.py

```python
import timeit
import functools
import utilities


def NN(distances, initial_vertex, vertices):
    N = len(vertices)

    circuit = [initial_vertex]
    current_vertex = initial_vertex

    unvisited_vertices = list(vertices)
    unvisited_vertices.remove(initial_vertex)

    cost = 0
    for i in range(N - 1):
        next_vertex = unvisited_vertices[0]
        min_distance = distances[current_vertex][next_vertex]
        for vertex in unvisited_vertices:
            if distances[current_vertex][vertex] < min_distance:
                next_vertex = vertex
                min_distance = distances[current_vertex][vertex]
        circuit.append(next_vertex)
        unvisited_vertices.remove(next_vertex)
        current_vertex = next_vertex
        cost = cost + min_distance

    circuit.append(initial_vertex)
    cost = cost + distances[current_vertex][initial_vertex]
    return circuit, cost


if __name__ == "__main__":
    # build our moons array(our entry point is also considered a moon)
    moons = utilities.read_input('input.txt')

    # build our 2-D matrix with the distances
    distances = utilities.calc_distances(moons)

    # run the nearest neighbour algorithm
    circuit, cost = NN(distances, moons[0], moons)

    # calculate average running speed of NN for our problem
    t = timeit.Timer(functools.partial(NN, distances, moons[0], moons))
    timeit_results = t.autorange()

    # print the results from NN
    utilities.print_alg_results("nearest neighbour", distances, circuit, cost,
    ↪  timeit_results)
```

A.3 Brute Force

bruteforce.py

```python
import timeit
import functools
import itertools
import utilities


def BF(distances, initial_vertex, vertices):
    unvisited_vertices = list(vertices)
    unvisited_vertices.remove(initial_vertex)

    # enter the starting and ending point of our circuit
    min_circuit = [initial_vertex, initial_vertex]
    # create our initial circuit in order to have a min
    min_circuit[1:1] = unvisited_vertices
    # find its cost
    min_cost = 0
    for source, destination in zip(min_circuit, min_circuit[1:]):
        min_cost = min_cost + distances[source][destination]

    # create all permutations and test them
    permutations_iter = itertools.permutations(unvisited_vertices)
    for permutation in permutations_iter:
        circuit = [initial_vertex, initial_vertex]
        circuit[1:1] = permutation
        cost = 0
        for source, destination in zip(circuit, circuit[1:]):
            cost = cost + distances[source][destination]
        if cost < min_cost:
            min_circuit = circuit
            min_cost = cost
    return min_circuit, min_cost


if __name__ == "__main__":
    # build our moons array(our entry point is also considered a moon)
    moons = utilities.read_input('input.txt')

    # build our 2-D matrix with the distances
    distances = utilities.calc_distances(moons)

    # run the brute force algorithm
    circuit, cost = BF(distances, moons[0], moons)

    # calculate average running speed of BF for our problem
    t = timeit.Timer(functools.partial(BF, distances, moons[0], moons))
    timeit_results = t.autorange()

    # print the results from NN
    utilities.print_alg_results("brute force", distances, circuit, cost,
    ↪  timeit_results)
```

A.4 Complexity Experiment

analysis.py

```python
from bruteforce import BF
from nearestneighbour import NN
from utilities import calc_distances, Moon
from string import ascii_lowercase
from decimal import Decimal
import random
import timeit
import functools
import sys


def print_analysis(results):
    # redirect all prints to our file
    sys.stdout = open("analysis_output.txt", "w")

    print("Analysis Results")
    print("-" * len("Analysis Results"))
    print()
    print("n: number of vertices")
    print("NN: average time(seconds) in order to complete the nearest neighbour
    ↪  algorithm")
    print("NN trials: number of times the algorithm was run for n")
    print("BF: average time(seconds) in order to complete the brute force
    ↪  algorithm")
    print("BF trials: number of times the algorithm was run for n")
    print()

    print('{:3s}'.format("n"), end="|", flush=True)
    print('{:>10s}'.format("NN"), end="|", flush=True)
    print('{:>10s}'.format("NN trials"), end="|", flush=True)
    print('{:>10s}'.format("BF"), end="|", flush=True)
    print('{:>10s}'.format("BF trials"), end="|\n", flush=True)

    for result in results:
        print('{:<3d}'.format(result["n"]), end="|", flush=True)
        print('{:10.2E}'.format(Decimal(result["NN"])), end="|", flush=True)
        print('{:10d}'.format(result["NN trials"]), end="|", flush=True)
        print('{:10.2E}'.format(Decimal(result["BF"])), end="|", flush=True)
        print('{:10d}'.format(result["BF trials"]), end="|\n", flush=True)


if __name__ == "__main__":
    random.seed(1)
    results = []

    for N in range(2, 13):
        moons = []
        result = {"n": N}
        for i in range(N):
            moon = Moon(ascii_lowercase[i], random.randint(1,13000))
            moons.append(moon)

        distances = calc_distances(moons)

        NN_timer = timeit.Timer(functools.partial(NN, distances, moons[0], moons))
        NN_results = NN_timer.autorange()
        result["NN"] = NN_results[1]/NN_results[0]
        result["NN trials"] = NN_results[0]

        BF_timer = timeit.Timer(functools.partial(BF, distances, moons[0], moons))
        BF_results = BF_timer.autorange()
        result["BF"] = BF_results[1]/BF_results[0]
        result["BF trials"] = BF_results[0]

        results.append(result)

    print_analysis(results)
```

References

[1] I. Ridpath, *A Dictionary of Astronomy*, 2 edition. Oxford ; New York: Oxford University Press, 20th Feb. 2012, 544 pp., ISBN: 978-0-19-960905-5.

[2] A. McMahon, ‘Space exploration, a continuing effort’, *GEOPHYSICS*, vol. 26, no. 3, pp. 374–393, 1st Jun. 1961, ISSN: 0016-8033. DOI: 10.1190/1.1438883.

[3] O. Grasset, M. K. Dougherty, A. Coustenis, E. J. Bunce, C. Erd, D. Titov, M. Blanc, A. Coates, P. Drossart, L. N. Fletcher, H. Hussmann, R. Jaumann, N. Krupp, J. .-.-P. Lebreton, O. Prieto-Ballesteros, P. Tortora, F. Tosi and T. Van Hoolst, ‘JUpiter ICy moons explorer (JUICE): An ESA mission to orbit ganymede and to characterise the jupiter system’, *Planetary and Space Science*, vol. 78, pp. 1–21, 1st Apr. 2013, ISSN: 0032-0633. DOI: 10.1016/j.pss.2012.12.002.

[4] T. Greicius. (16th Jun. 2015). Europa overview, NASA, [Online]. Available: http://www.nasa.gov/europa/overview/index.html (visited on 28/04/2018).

[5] D. R. Williams. Jovian satellite fact sheet, [Online]. Available: https://nssdc.gsfc.nasa.gov/planetary/factsheet/joviansatfact.html (visited on 02/05/2018).

[6] Mission juno, Mission Juno, [Online]. Available: https://www.missionjuno.swri.edu/ (visited on 02/05/2018).

[7] NASA’s eyes: Eyes on juno, NASA’s Eyes, [Online]. Available: http://eyes.nasa.gov/eyes-on-juno.html (visited on 02/05/2018).

[8] K. H. Rosen, *Discrete Mathematics and Its Applications Seventh Edition*, 7th edition. New York: McGraw-Hill Education, 14th Jun. 2011, 1072 pp., ISBN: 978-0-07-338309-5.

[9] J. J. Sylvester. (7th Feb. 1878). Chemistry and algebra, Nature, [Online]. Available: https://www.nature.com/articles/017284a0 (visited on 19/05/2018).

[10] W. T. Tutte, *Graph theory*. Addison-Wesley Pub. Co., Advanced Book Program, 1984, 368 pp., Google-Books-ID: pLwdAQAAMAAJ, ISBN: 978-0-201-13520-6.

[11] R. Gera, S. Hedetniemi and C. Larson, Eds., *Graph Theory: Favorite Conjectures and Open Problems - 1*, Problem Books in Mathematics, Springer International Publishing, 2016, ISBN: 978-3-319-31938-4.