**Discrete Mathematics Project.**

**By George Katsonis.**

**C1. Optimization Algorithms:**

Finding an optimized route between the vertices of a graph has always been a very important problem, as it allows to model and optimize a very large variety of problems. As such, several algorithms have been developed to try and find this path while being as efficient as possible. The following are some of the most important general use algorithms for this purpose, that apply on directed graphs:

* Dijkstra's algorithm
* Bellman-Ford algorithm
* Nearest neighbor algorithm
* A\* search algorithm
* Floyd-Warshall algorithm
* Brute Force

Undoubtedly, Dijkstra’s algorithm is the most historically significant and widely used algorithm of this list; however, due to the pragmatic simplification of the project and since we seek a Hamiltonian path, the most efficient approach is to use a modified version of the Shortest neighbor algorithm.

The nearest neighbor search (NNS) surfaced during the 70s as the answer to Donald Knuth’s post-office problem, that aimed to assign residences to the nearest post office. As the name suggests, the basic idea of this algorithm is to identify the point of a set, that is the closest to a predefined starting position.

Building upon this basic idea, many iterations of this algorithm have been developed over the years, with a myriad of applications on a multitude of scientific fields, like pattern recognition, DNA sequencing, Plagiarism detection and, most importantly, AI and machine learning. Depending on the variation the complexity of the algorithm changes, but for our implementation we have O(V\*E) complexity where V is the number of vertices and E the number of edges.

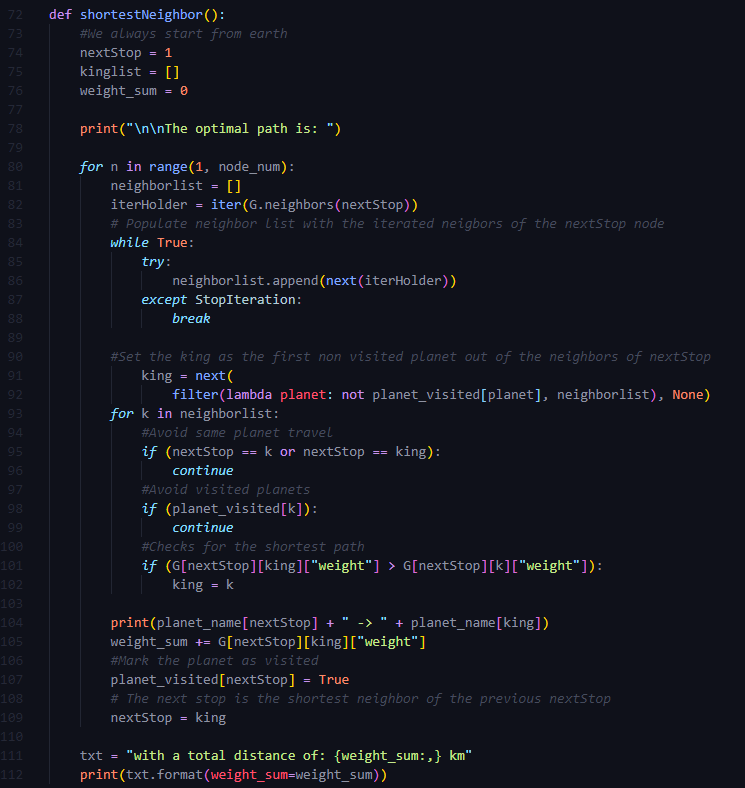
Depending on the application, NNS has adapted to accommodate any number set S of N data in a d-dimensional space; however, such sophisticated tools are beyond the scope and needs of this project. The variation applied here was created from the ground up with only the basic idea of NNS in mind, to be applicable on projects of the same nature, but to also be flexible enough to work with any number of destinations.

It is important to note that NNS is a greedy search algorithm, meaning it favors short term decision making, rather than long term. This means that for more complicated systems, the provided solution might not be the optimal one.

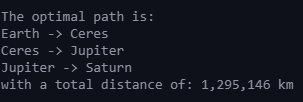
**C2. Applying the Algorithm:**

NNS is applied to calculate the optimal route, when considering distance. In the case that we assume the spacecraft will move with a constant velocity, and we do not calculate the time spend on the surface of each celestial body, calculating the time needed for the expedition would be a simple multiplication; however, in realistic terms the best we could do is very roughly approximate the total time by taking such factors into consideration. Thus, calculating the distance is more appropriate as it stands as an independent measurement that can be used in further calculations. Text version of the code can be found in Appendix 1.

Python Code:



Output:



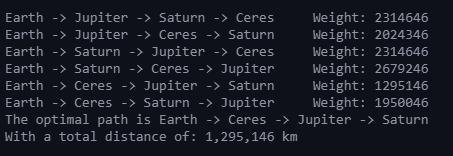
**C3. Finding the Optimal Solution:**

Due to the simplifications of the project (mentioned in part B), the brute force algorithm was very simple to develop and executed very fast. However, since another layer of nesting would be required for each added node, it is clear how fast it would increase the time complexity of the algorithm. Text version of the code can be found in Appendix 2.

Python code:



Output:



As explained in part B, the order in which the moons will be visited after landing on the planet does not matter in terms of total distance. Furthermore, we want to end up on Titan instead of Triton since Titan is highly habitable and an excellent final stop for the trading route. Thus, we will choose the suggested path based on moon radius, as to maximize material gathering between trips to convert to fuel. The suggested path is as follows: Earth 🡪 Ceres 🡪 Jupiter 🡪 Ganymede 🡪 Callisto 🡪 Europa 🡪 Io 🡪 Saturn 🡪 Triton 🡪 Titan.

**C4. Evaluation:**

The NNS algorithm extremely efficient compared to brute force for two major reasons, time complexity and flexibility. As stated above the time complexity for NNS is O(V\*E), while brute force is O(V-1!), making it a lot slower in comparison.

When it comes to flexibility, the brute force algorithm needs to be rewritten for every new node added since we need to compensate with more nested loops, while the NNS can be applied for any number of vertices and edges of any weight.

While the project tries to be as realistic as possible when it comes to the conditions and obstacles in such a travel, it is apparent that an actual endeavor of this magnitude would require hundreds other factors to be considered and included in any planning algorithms. Thus, all the tools in such case would be extremely sophisticated; however, it is very much possible that they would build upon these basic foundations and ideas.

If there were no time constraints, this project would benefit from a GUI application that would visualize the algorithms in real time, the use of more sophisticated tools as a showcase, and a framework that can accept any number of user defined destinations and distances that would then model and solve the problem with those parameters. Finally, from a pragmatic standpoint, factors such as gravitational slingshots and electromagnetic interferences would also be major concerns that would have to be considered.

**C5. Future Trajectories:**

As I am very interested in bioengineering and molecular medicine, one of the most interesting applications for me is the application of graph theory to model and emulate the interactions between proteins in the body, and identify their usage using AI.

In general, the use of AI and neural networks makes extensive use of graph theory since many AI models operate as networks of information.

More specifically an interesting application I drafted a paper on is the application of machine learning to determine the coefficients that lead to the characterization of an age-related protein, by analyzing large databases of age related proteins and testing for a large variety of proteinic features.

**Appendix 1:**

*#Nearest neighbor method*

def shortestNeighbor():

*#We always start from earth*

    nextStop = 1

    kinglist = []

    weight\_sum = 0

    print("\n\nThe optimal path is: ")

*for* n in range(1, node\_num):

        neighborlist = []

        iterHolder = iter(G.neighbors(nextStop))

*# Populate neighbor list with the iterated neigbors of the nextStop node*

*while* True:

*try*:

                neighborlist.append(next(iterHolder))

*except* StopIteration:

*break*

*#Set the king as the first non visited planet out of the neighbors of nextStop*

            king = next(

                filter(lambda planet: not planet\_visited[planet], neighborlist), None)

*for* k in neighborlist:

*#Avoid same planet travel*

*if* (nextStop == k or nextStop == king):

*continue*

*#Avoid visited planets*

*if* (planet\_visited[k]):

*continue*

*#Checks for the shortest path*

*if* (G[nextStop][king]["weight"] > G[nextStop][k]["weight"]):

                king = k

        print(planet\_name[nextStop] + " -> " + planet\_name[king])

        weight\_sum += G[nextStop][king]["weight"]

*#Mark the planet as visited*

        planet\_visited[nextStop] = True

*# The next stop is the shortest neighbor of the previous nextStop*

        nextStop = king

    txt = "with a total distance of: {weight\_sum:,} km"

    print(txt.format(weight\_sum=weight\_sum))

**Appendix 2:**

def bruteForce():

    minN = 0

    minK = 0

    minL = 0

    king = 0

*#Brute force method.*

*for* n in range(2, node\_num + 1):

*for* k in range(1, node\_num + 1):

*if* (n == k):

*continue*

*if* (k == 1):

*continue*

*for* l in range(1, node\_num + 1):

*if* (n == k or n == l or k == l):

*continue*

*if*(l == 1):

*continue*

*try*:

                    total\_weight = G[1][n]["weight"] + G[n][k]["weight"] + G[k][l]["weight"]

*except*:

*continue*

*if* (total\_weight < king):

                    king = total\_weight

                    minN = n

                    minK = k

                    minL = l

*#Combination Unique to the first iteration.*

*elif* (n == 2 and k == 3 and l == 4):

                    king = total\_weight

                    minN = n

                    minK = k

                    minL = l

                print(planet\_name[1] + " -> " + planet\_name[n]

                + " -> " + planet\_name[k] + " -> " + planet\_name[l] + "     Weight: " + str(total\_weight))

    print("The optimal path is " + planet\_name[1] + " -> " + planet\_name[minN]

        + " -> " + planet\_name[minK] + " -> " + planet\_name[minL])

    txt = "With a total distance of: {king:,} km"

    print(txt.format(king = king))

**References:**

Ahuja, R., Mehlhorn, K., Orlin, J. and Tarjan, R., 1990. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2), pp.213-223.

Arya, S., Mount, D., Netanyahu, N., Silverman, R. and Wu, A., 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6), pp.891-923.

Kerepesi, C., Daróczy, B., Sturm, Á., Vellai, T. and Benczúr, A., 2018. Prediction and characterization of human ageing-related proteins by using machine learning. *Scientific Reports*, 8(1).

Mahapatra, R. and Chakraborty, P., 2015. Comparative Analysis of Nearest Neighbor Query Processing Techniques. *Procedia Computer Science*, 57, pp.1289-1298.