380CT - Theoretical Aspects of Computer Science

Coursework Portfolio

Name: Adrian Raszkiewicz
StudentID: 7128671
Github: https://github.coventry.ac.uk/380CT-1819JANMAY/380CT-raszkiea
Group Members: Adrian Raszkiewicz (SID:7128671) and Osama Asim (SID:6676872)

Notation

Let G be a complete weighted graph with N vertices with the cycle length k

- **Vertices:** fundamental unit of which graphs are formed
- **Edges:** lines/paths connecting the different vertices of a graph
- **Directed Graph:** set of vertices connected together, where all the edges are directed and therefore unidirectional
- **Undirected Graph:** set of vertices connected together, where all the edges are bidirectional
- **Complete:** each pair of graph vertices is connected by an edge, the graph is also undirected and has no self-loops
- **Weighted:** each edge is given a numerical value (usually positive number), this is referring to the cost of the edge
- **Unweighted:** each edge is not given a numerical value
- **Cycle:** a path of edges and vertices where a vertex is reachable from itself
- **Total cost of the cycle:** sum of the edge weights of the cycle

Background

Traveling Salesman Problem (TSP)

Statement of the Problem

- **Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.**

This is a problem in graph theory requiring the most efficient Hamiltonian cycle, for the purpose of the most efficient path a salesman can take through each location. The Hamiltonian cycle is a graph cycle that visits each vertex exactly once.

This is one of the most intensively studied problems in optimization firstly formulated in 1930. The TSP has many important and exciting applications in real life, such as: logistics, scheduling and packing.

Therefore: Given an undirected complete graph $G(N)$ find an optimal tour (Hamiltonian cycle).

Criteria to consider:

- Minimize tour length
- Minimize mean arrival time at vertices ***(Jünger, M et al, 1995)***

Complexity Classification

Optimization TSP

- Given G, find a cycle of minimal total cost.

No general solution is available for this problem, it is considered a NP-Hard problem.

It is considered an NP-Hard problem (nondeterministic polynomial time) as there is no known polynomial algorithm that can solve it, so the problem will grow exponentially with the problem size. It also means that the algorithm for this problem can be translated into one for solving any NP-problem, therefore this problem is at least as hard as any NP-problem but could be harder. In the same way, it also means that any problem in NP can be polynomially reduced to this NP-hard problem ***(Sipser, M.V, 2018)***.

There has been a great deal of confusion about the TSP being NP-complete. I will now explain why it is in fact not NP-complete with regards to optimisation.

- NP: problems that their solutions can be verified in polynomial time
- NP-hard: problems that can't be solved in polynomial time
- NP-complete: problems that are both NP and NP-hard

TSP optimization is not NP-complete as it is not NP. NP solutions need to be solvable in a polynomial time(time taken for computer to solve the problem) and as we know this problem has not yet been solved regarding that sense.

Lets assume we have found the solution for the shortest routes across all the edges of a graph, and therefore the shortest route for our salesperson. There is no way we can possibly verify that the solution we have given is the shortest loop, which means that this problem cannot be solved in polynomial time and therefore is not NP. The only way we can actually verify that this is the shortest route would be to solve the TSP, this not only takes exponential time meaning it cannot be checked through polynomial time, but also hasn't ever been done before ***(Evan Klitzke, 2017)***.

Decisional TSP

- Given a total cost k, decide if G has a cycle of length at most k.

As there is no known **best** polynomial-time algorithm for solving the TSP, we can change this into a 'decision instance' of the problem which makes the problem far more simpler.

The decisional instance of the TSP problem is NP-complete as it is in both NP and NP-hard.

Here we are determining whether the weight of a solution in graph G is at most the cost of k. This certainly can be verified in Polynomial time. This is a yes or no decision as we are given k. So we are checking for a specific value ***(Marianne Freiberger, 2012)***.

Search TSP

- Given a total cost k, decide if G has a cycle of length k, if it exists.

This is a similar instance to the decisional problem, however this time the answer will not be either 'yes' or 'no' as we are not focusing on just finding a solution that is smaller than k. Instead we are actually searching for the graph and finding k.

So once we are given a complete weighted graph G, it is only a matter of time and computation to locate the most efficient path. Therefore the search instance of the TSP problem also belongs to the class of NP-complete problems ***(Jünger, M et al, 1995)***.

Exhaustive Search

Exhaustive Search known also as Brute Force Search is a search algorithm that tests each possible solution sequentially to determine the best one, this programming style does not include any shortcuts to improve performance but instead focuses on the best solution.

While Exhaustive Search has been proven to be easily to implemented and is guaranteed to find the 'best' solution to a problem; it's cost is directly proportional to the number of solutions available which is where the impracticality of the Exhaustive Search stems from when considering more real world problems.

Consider the simple fact that it will check through all possible solutions, this will create a search tree that extends exponentially as the problem size increases and therefore hits the scalability wall. This phenomenon is called the 'combinatorial explosion' and it refers to the rapid growth of the complexity of a problem when it is affected by its inputs or constraints.

Exhaustive search suffers from the following scalability issues:

- It scales horribly performance (time) wise
- It scales horribly memory wise

The time complexity for this is $O(n*m)$.
This means that if we were to search for a string of **n** characters in a string of **m** characters in would take us n*m tries.

Therefore, Exhaustive Search should only be applied to problems that have either: a limited problem size or problems to which the simplicity of the implementation is more important than the speed ***(Geoffrey De Smet et al)***.

Methodology

Generation of random instances

A random graph is a graph of which the properties such as the graph vertices or the graph edges are determined by random.

The random graph is generated with the inbuild python library NetworkX, using the erdos_renyi_graph function. This function takes the parameters:

- 'Nodes' for the number of vertices, as a integer
- 'ProbabilityCnct' for the probability of city connection, as a integer (1 = 100% probability of connection)
- 'directed' for the nature of the graph edges, as a Boolean

The graph direction was set to false as we require an undirected graph.
The erdos_renyi_graph function doesn't allow for a weighted graph; therefore the code was adapted to return random weights for the paths.
The code was also adapted to return different instances of cities N and different weights between them.

In [1]:
```
import networkx as nx
import scipy as sp
import random
from random import randint
```

In [2]:
```
#Function for random generation of a directed graph with specified number of nodes
#and probability of connection between nodes
def RandomGraphGenerator(Nodes, ProbabilityCnct):

    G = nx.erdos_renyi_graph(Nodes, ProbabilityCnct, directed=False)

    for (u, v) in G.edges():
        G.edges[u,v]['weight'] = random.randint(1,7)
    return(G)

#Plotting the randomly generated Graph
if __name__ == "__main__":
    for i in range (1,5):
        Random_Graph = RandomGraphGenerator((random.randint(8,13)), 1)
        A = nx.adjacency_matrix(Random_Graph)
        print(A.todense())
    nx.draw_networkx(Random_Graph, node_color="green", edge_color="red")
```

```
[[0 6 7 6 4 2 6 5 4 7 7 5 1]
 [6 0 2 3 2 6 1 6 3 3 5 4 6]
 [7 2 0 6 3 3 3 5 3 1 7 1 1]
 [6 3 6 0 3 4 4 6 2 5 3 4 5]
 [4 2 3 3 0 3 2 2 6 3 5 6 7]
 [2 6 3 4 3 0 5 5 7 6 7 2 7]
 [6 1 3 4 2 5 0 4 4 5 1 6 5]
 [5 6 5 6 2 5 4 0 7 1 2 6 2]
 [4 3 3 2 6 7 4 7 0 1 6 1 1]
 [7 3 1 5 3 6 5 1 1 0 4 6 3]
 [7 5 7 3 5 7 1 2 6 4 0 2 6]
 [5 4 1 4 6 2 6 6 1 6 2 0 6]
 [1 6 1 5 7 7 5 2 1 3 6 6 0]]
[[0 3 5 2 7 4 3 5 6 5 7 4 5]
 [3 0 2 1 7 5 5 4 3 4 3 5 2]
 [5 2 0 1 1 5 6 5 6 3 5 4 1]
 [2 1 1 0 3 2 1 3 1 1 4 2 2]
 [7 7 1 3 0 1 7 2 6 2 6 1 1]
 [4 5 5 2 1 0 6 2 2 4 5 7 5]
 [3 5 6 1 7 6 0 5 5 7 7 5 7]
 [5 4 5 3 2 2 5 0 2 5 7 2 3]
 [6 3 6 1 6 2 5 2 0 6 1 5 4]
 [5 4 3 1 2 4 7 5 6 0 4 3 7]
 [7 3 5 4 6 5 7 7 1 4 0 6 5]
 [4 5 4 2 1 7 5 2 5 3 6 0 5]
 [5 2 1 2 1 5 7 3 4 7 5 5 0]]
[[0 5 6 4 4 5 6 2 2 6 1 2]
 [5 0 2 6 5 4 7 3 3 4 5 7]
 [6 2 0 4 3 4 5 7 6 2 3 4]
 [4 6 4 0 2 1 2 2 3 5 2 4]
 [4 5 3 2 0 4 7 2 4 6 7 7]
 [5 4 4 1 4 0 5 3 5 4 5 7]
 [6 7 5 2 7 5 0 5 7 5 2 6]
 [2 3 7 2 2 3 5 0 6 1 5 3]
 [2 3 6 3 4 5 7 6 0 5 4 7]
 [6 4 2 5 6 4 5 1 5 0 2 5]
 [1 5 3 2 7 5 2 5 4 2 0 5]
 [2 7 4 4 7 7 6 3 7 5 5 0]]
[[0 4 4 1 2 1 4 5 6 6 6]
 [4 0 2 2 2 2 1 3 2 1 5]
 [4 2 0 6 2 2 7 1 1 3 6]
 [1 2 6 0 3 3 3 1 3 4 5]
 [2 2 2 3 0 2 4 2 2 1 1]
 [1 2 2 3 2 0 3 5 7 3 6]
 [4 1 7 3 4 3 0 4 3 6 7]
 [5 3 1 1 2 5 4 0 4 1 4]
 [6 2 1 3 2 7 3 4 0 6 2]
 [6 1 3 4 1 3 6 1 6 0 7]
 [6 5 6 5 1 6 7 4 2 7 0]]
```

```
C:\Users\44770\AppData\Local\Programs\Python\Python37-32\lib\site-packages\networkx\drawing\nx_pylab.py:611: MatplotlibDeprecationWarning: isinstance(..., numbers.Number)
  if cb.is_numlike(alpha):
```

Experiments Both the 'Greedy Search' and the 'Greedy Randomized Adaptive Search Procedure' will firstly be tested on their own with a different number of vertices to evaluate their space and time complexity with an increasing problem size.

The experiments will be conducted with a large input size (vertices) to make sure I'm testing the implementation over a large enough sample.

To ensure the algorithms are tested correctly:

- Large enough input sample variation
- Compared with the same number of nodes + weights between them
- Make sure this is an undirected, connected, weighted graph for each instance
- Work out the TSP firstly by hand, compare with result from code

Theory

Greedy Search

The Greedy Search is a very simple algorithm that is used for optimization problems. This algorithm makes the best local choice available at each step of computation and hopes to arrive at the optimal solution to a problem. The result solution is usually feasible, however not always guaranteed to be the optimal solution.

Pseudocode

1. $optimalroute \leftarrow \emptyset$
2. $optimalweight \leftarrow \infty$
3. **for all** permutations nodes N in G **do**
4. $route.append(NextNode)$
5. $weight \leftarrow 0$
6. $NextNode \leftarrow$ closest nodes to $StartNode$
7. $weight \leftarrow weight +$ Distance to $NextNode$
8. $route.append(NextNode)$
9. $while$ weight of $route < optimalroute$ **do**
10. $NextNode \leftarrow$ closest nodes to $CurrentNode$
11. $weight \leftarrow weight +$ Distance to $NextNode$
12. $route.append(NextNode)$
13. **if** $weight < optimalweighth :$
14. $optimalweight \leftarrow weight$
15. $optimalroute \leftarrow route$
16. **return** $optimalroute, optimalweight$

Parameters

This pseudocode takes one input value G which is a graph that has nodes (cities) inside as N

It returns two values:

- $optimalroute$ = **set** of **integers**
- $optimalweight$ = **integer**

Explanation of pseudocode

1. Consider the first node in the graph as the start point
2. Iterate over all nodes N in graph G
3. Consider the next closest node and append the distance weight from current node to the next closest node
4. while the distance weight from step 3 is smaller than optimal route DO:

- Consider the next closes node and append the distance weight from the start node to the next closest node
- when weight of current route is bigger than optimal route break out from while loop

1. if weight of route is smaller than the optimal weight DO:
2. new optimal weight is the weight found and optimal route is the route corresponding to that weight
3. go back to step 1 and consider the second node in the graph as the starting point, repeat this until all nodes are exhausted as starting point (N iterations)
4. return optimal weight and optimal route K

Time Complexity

The time complexity of the above algorithm is $O(n^2)$ as it's performance is directly proportional to the square of the size of the input data set.
There is a nested while loop inside a for loop.
The for loop will iterate N times over G, and the nested while loop will iterate n-1 times inside the for loop for each route, minus the node that has already been traversed.
Therefore, this algorithm involves nested iterations over the data set.

Greedy Randomized Adaptive Search Procedure (GRASP) - Metaheuristic

THe GRASP method is composed of iterations over successive iterations of the Greedy solutions and the improvements of these using local search to find the most optimal one.

Pseudocode

GRASPsearch()

$BestSolution \leftarrow \emptyset$

1. **while** stop condition is not met, **do**
2. $path < -GreedySearch$
3. $newpath < -LocalSearch(path)$
4. $if\ (cost(newpath) is better than Cost(BestSolution)$
5. $BestSolution < -newpath$
6. end
7. end
8. **return** $BestSolution$

Paramaters

This pseudocode takes one input value **GreedySearch** which are the results from the Greedy Search on graph G.

It returns one value $BestSolution$ = best path found in G

Explanation of pseudocode

1. Initialize the GRASP function with an empty $BestSolution$
2. while the stop condition is not met, do:
3. Apply Greedy Algorithm to graph G and store results in $path$
4. Apply $LocalSearch$ to $path$, store results in $newpath$
5. if cost of $newpath$ is more optimal than curent $BestSolution$
6. store $newpath$ inside $BestSolution$
7. return $BestSolution

- the stop conition could be the number of permutations defined by the user
- this algorithm calls both the $LocalSearch$ and $GreedySearch$

Time Complexity
The time complexity of that above aloright is $0(N)$, where N is the order of edges along the shortest path from a starting point. There is only one while loop which iterates over all permutations, therefore it is $O(N)$ as it iterates N times, where N is specified by the user. This however is the case only when we look at this algorithm individually.

If we consider that this algorithm calls two other algorithms $LocalSearch$ and $GreedySearch$, the time complexity will be severly affected as there will be more nested loops and intertions etc.

Code to mimic the nature of O(n) and O(n^2)

In [3]:
```
# code adapted from https://stackabuse.com/big-o-notation-and-algorithm-analysis-with-python-examples/?fbclid=IwAR2qrTfAyvIcharG25oC_Nw1-31j7u8sJA9u8P9uBnOHaTT1fQQ1DDP11UU
import matplotlib.pyplot as plt

#Initialise lists
X_Coordinates = []
Exh_Grdy = []
Grsp = []

#Generate X cordinates from 0 to 40 with increments of 2
for i in range (0,40,2):
    X_Coordinates.append(i)

#Generate square of X_coordinates for O(n)square and minute increase for O(n)
for i in X_Coordinates:
    squared = i*i
    Exh_Grdy.append(squared)
    Grsp.append(i*1.0002)

plt.plot(X_Coordinates,Exh_Grdy, color='orange', linewidth=3,
         label = 'O(N^2) - Exhaustive and Greedy')
plt.plot(X_Coordinates,Grsp, color='green', linewidth=3,
         label = 'O(N) - GA & GRASP - (Meta)')
plt.legend(loc='best')
```

Out[3]: <matplotlib.legend.Legend at 0xe7fc050>

Practice

Greedy Search

In [4]:
```
#code adapted from https://stackoverflow.com/questions/41749254/basic-greedy-search-in-python
start = time.time()
List=[]
def basic_greedy(n):
    # greedy search algorithm
    d_dict = List=[]
def basic_greedy(n):
    # greedy search algorithm
    d_dict = {0: [(0, 0), (1, 1), (2, 1), (3,1), (4,1)], 1: [(0, 1), (1, 0), (2, 2), (3,2), (4,2)], 2: [(0, 1), (1, 2), (2, 0), (3,3), (4,3)], 3: [(0, 1), (1, 2), (2, 3), (3,0), (4,4)], 4: [(0, 1), (1, 2), (2, 3), (3,4), (4,0)]}  # dict of lists of tuples such that nodei : [ (neighbourj, distancej), .... ]
    currentCity = n
    tour = []   # list of covered nodes
    tour.append(currentCity)
    distanceTravelled = 0   # distance travelled in tour
    while len(set([neighbourCity for (neighbourCity, distance) in d_dict.get(currentCity, [])]).difference(set(tour))) > 0:  # set(currentCityNeighbours) - set(visitedNodes)
        # way 1 starts
        minDistanceNeighbour = None
        minDistance = None
        for eachNeighbour, eachNeighbourdDistance in d_dict[currentCity]:
            if eachNeighbour != currentCity and eachNeighbour not in tour:
                if minDistance is not None:
                    if minDistance > eachNeighbourdDistance:
                        minDistance = eachNeighbourdDistance
                        minDistanceNeighbour = eachNeighbour
                else:
                    minDistance = eachNeighbourdDistance
                    minDistanceNeighbour = eachNeighbour
        nearestNeigbhourCity = (minDistanceNeighbour, minDistance)
        # way 1 ends
        # way 2 starts
        # nearestNeigbhourCity = min(d_dict[currentCity], key=lambda someList: someList[1] if someList[0] not in tour else 1000000000)  # else part returns some very large number
        # way 2 ends
        tour.append(nearestNeigbhourCity[0])
        currentCity = nearestNeigbhourCity[0]
        distanceTravelled += nearestNeigbhourCity[1]
    print(tour)
    print(distanceTravelled)
    List.append(distanceTravelled)
    List.sort()
    print("Best Distance:", *List[:1])

def best_greedy():
    for i in range (0,5):
        basic_greedy(i)  # dict of lists of tuples such that nodei : [ (neighbourj, distancej), .... ]
    currentCity = n
    tour = []   # list of covered nodes
    tour.append(currentCity)
    distanceTravelled = 0   # distance travelled in tour
    while len(set([neighbourCity for (neighbourCity, distance) in d_dict.get(currentCity, [])]).difference(set(tour))) > 0:  # set(currentCityNeighbours) - set(visitedNodes)
        # way 1 starts
        minDistanceNeighbour = None
        minDistance = None
        for eachNeighbour, eachNeighbourdDistance in d_dict[currentCity]:
            if eachNeighbour != currentCity and eachNeighbour not in tour:
                if minDistance is not None:
                    if minDistance > eachNeighbourdDistance:
                        minDistance = eachNeighbourdDistance
                        minDistanceNeighbour = eachNeighbour
                else:
                    minDistance = eachNeighbourdDistance
                    minDistanceNeighbour = eachNeighbour
        nearestNeigbhourCity = (minDistanceNeighbour, minDistance)
        # way 1 ends
        # way 2 starts
        # nearestNeigbhourCity = min(d_dict[currentCity], key=lambda someList: someList[1] if someList[0] not in tour else 1000000000)  # else part returns some very large number
        # way 2 ends
        tour.append(nearestNeigbhourCity[0])
        currentCity = nearestNeigbhourCity[0]
        distanceTravelled += nearestNeigbhourCity[1]
    print(tour)
    print(distanceTravelled)
    List.append(distanceTravelled)
    List.sort()
    print("Best Distance:", *List[:1])

def best_greedy():
    for i in range (0,5):
        basic_greedy(i)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-488ad4c5b466> in <module>
      1 #code adapted from https://stackoverflow.com/questions/41749254/basic-greedy-search-in-python
----> 2 start = time.time()
      3 List=[]
      4 def basic_greedy(n):
      5     # greedy search algorithm

NameError: name 'time' is not defined
```

In [ ]:
```
import time
start = time.time()
best_greedy()
end = time.time()
print("Execution time: ",(end - start))
```

Reflection

Development

The core idea for the pseudocode for **Greedy Search** and **GRASPsearch** I have explored together with Osama Asim, however the implementation and explanation was done individually.

In order to implement the pseudocode for **Greedy Search** I together with Osama Asim have used code proposed by a user on stackoverflow.com, we believe that this algorithm was well developed from the knowledge we gained. We have also adapted the code as it was not suitable for our problem.

We have added code to do the following:

- return the best solution from the paths calculated
- return the execution time of the code

Unfortunately we didn't get enough time to try to implement the **GRASPsearch** in code, only in the pseudocode.

Knowledge gained and its application

1. **Experience with Jupyter Notebook** I have had my first experiences with Jupyter Notebook thanks to this assignment. This is a very powerful tool which allows for features like: running live code along text, equations in their proper format and various forms of visualization. I believe this is very convenient when presenting work that includes: text, code, and visualization features as you have an interactive document that stores everything neatly in on place. Because of Jupyter I have also learned how to use both: Markdown which is a lightweight mark-up language and allows for formatting plane text with a specific syntax; and LaTeX mathematics which is another document preparation system that allows to format the output with a specific syntax and is usually done for mathematical representations
2. **Complexity Classification** Because of the fact that we were required to provide the complexity classifications, I have learned a great deal about this topic. It was introduced to use briefly in the previous year's however I believe that I didn't fully understand the concept, but instead more or less memorized the most popular algorithms and their O notation. Through the research I have done I can actually now understand what a specific Big O notation means for the algorithm, what effect it has on the time complexity with regards to the input and even how to calculate the time complexity from code.

Improvement area

1. **Background research** If I was to start over again, I would definitely make sure that I revised more literature about the problem. I have explored the minimal knowledge required and started attempting the coursework as soon as possible as I was stressing about my time management. I believe this was a hindrance towards my work as I didn't have enough knowledge on the field and kept on finding new facts about the problems I was working on, which caused a lot of trouble for me as I had to constantly update my pseudocode or just generally the idea behind any implementation or explanation throughout the report.
2. **Time Allocation** Time management is another factor I would definitely improve. I have started the coursework fairly early in March, so the problem was not just starting too late but the frequency at which I worked at this assignment. I would allow for more hours in my week to explore the problems stated in the assignment so that I would have more stress-free time to work on them. The reason behind this I believe was the fact that I'm in my final year and would really like to get the highest grades on all my assignments, therefore I was spending the same amounts of time on all of them through the week which was not a sensible thing to do as this deadline was the closest. Because of time management I have also not had time to implement the **GRASPsearch**, which is a shame as I would like to explore this as it is far more complex than the previous and does yield better results. Another factor affected by time management was the number of experiments I have done, if I had more time I could test the implementation over a larger sample to make them more accurate.

References

Evan Klitzke. (2017). The Traveling Salesman Problem Is Not NP-complete. Evan Klitzke's web log. [online] available at https://eklitzke.org/the-traveling-salesman-problem-is-not-np-complete [Accessed 18 Mar.2019]

Geoffrey De Smet, Jiri Locker, Christopher Chianelli and Musa Talluzi. (unknown). Chapter 8. Exhaustive Search. jboss. Community Documentation. [online] available at https://docs.jboss.org/drools/release/6.2.0.Beta3/optaplanner-docs/html/exhaustiveSearch.html [Accessed 28 Mar.2019]

Jünger, M., Reinelt, G. and Rinaldi, G. (1995). The traveling salesman problem. Handbooks in operations research and management science, 7, pp.225-330. [online] available at https://www.sciencedirect.com/science/article/pii/S0927050705801215 [Accessed 12 Mar.2019]

Sipser, M.V.(2018). Introduction to the Theory of Computation, Third International Edition. Ashford Colour Press Ltd. [Accessed 28 Mar.2019]

Appendix

TSP by hand: