NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ2001 Algorithms
Individual Assignment

20 November 2020

Done By: Low Yu Benedict
Matriculation Number: U1821762E
Tutorial/Lab Group: SSP2
Email: lowy0065@e.ntu.edu.sg

Question 1

Formally, two graphs are considered isomorphic if there is a bijection between the set of nodes in *G1* and *G2, f: V(G1) →V(G2)* such that any two nodes *u, v* are adjacent in *G1* and *f(u)* and *f(v)* are also adjacent in *G2* [1].

We can atomize the definition into two steps. Firstly, we map the set of nodes $n_1$in *G1* to the nodes n_2 in *G2.* Secondly, we check if the mapped nodes preserve the adjacency of both graphs, and thus confirm if *G1* and *G2* are isomorphic.

We start by mapping nodes in *G2* that have the same degree as nodes in *G1*.

Graph G1	Graph G2
a = {b, d}	d = {a, b}
b = {a, c, d}	a = {c, d, b}
c = {b, d}	c = {a, b}
d = {a, c, b}	b = {c, d, a}

By comparing the nodes based on their degree, we can map the following:

1) G1(d) → G2(b)
2) G1(b) → G2(a)
3) G1(a) → G2(d)
4) G1(c) → G2(c)

After matching nodes between *G1* and *G2*, an adjacency matrix is created for both G1 and G2.

G1	a	b	c	d
a	0	1	0	1
b	1	0	1	1
c	0	1	0	1
d	1	1	1	0

G2	d	a	c	b
d	0	1	0	1
a	1	0	1	1
c	0	1	0	1
b	1	1	1	0

As both adjacency matrixes are the same, we can confirm that every node in *G1* has a corresponding node mapped in *G2* that preserves the adjacency in *G1*. Therefore, G1 and G2 are isomorphic.

The graph isomorphism problem is considered an NP class problem. Given two graphs *G1* and *G2* as well as a certificate (i.e. a mapping of permutation done in Step 1) we can first verify if the certificate is a valid permutation of the graph; which in the case of a complete graph would result in a worst time of $O(n^2)$. After which, we can verify if the permutation of *G1* and *G2* are isomorphic in $O(n + v)$ time. As such, given two graphs and a certificate we can solve if they are isomorphic in $O(n^2)$, which is in polynomial time.

The graph isomorphism (GI) problem is likely not an NP-Complete class problem. GI has been accomplished in quasi-polynomial (QP) time [2]. The exponential time hypothesis (ETH) states that NP ⊄ QP [3] therefore GI is likely not an NP-complete class problem, and by extension also not an NP-Hard problem.

The graph isomorphism problem is also probably not a P class problem. It is noted that the Group Isomorphism problem can be reduced to a GI problem, but that the group isomorphism problem is solvable in $n^{O(log n)}$ which is not known to be in P class [2][4].

Ultimately, it seems like the classification of GI as NP-intermediate is still debated and I cannot be absolutely certain if GI is neither P nor NP-Complete. The papers I have read and cited suggests that GI is very likely an NP-intermediate class problem as of current research – belonging to NP but not in P or NP-Complete classes.

Question 2

Question 2 (a)

A possible approximation algorithm for the Traveling Salesman Problem (TSP) would be the Nearest Neighbour (NN) algorithm. As the name implies, the NN algorithm attempts to find the optimal route by choosing the nearest neighbour to its current node at each step of the graph traversal.

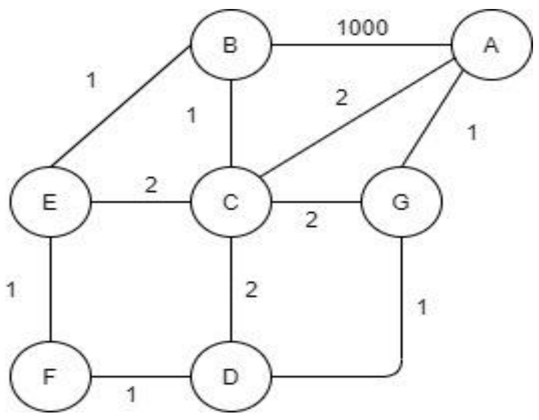
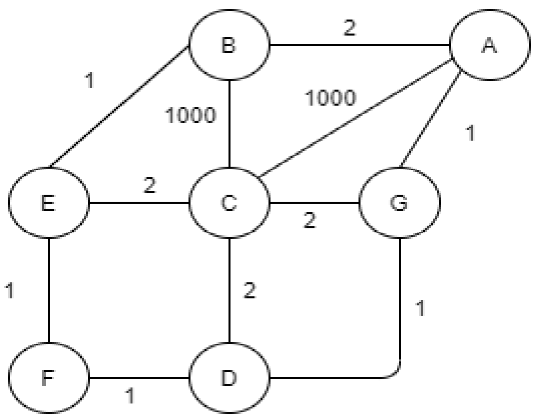
Finding the nearest neighbour takes $O(n)$, in the event of a complete graph. This occurs for *n* number of nodes, thus resulting in a time complexity of $\Theta(n^2)$.

One limitation of NN is its poor performance in Symmetric and Asymmetric TSP (STSP, ATSP) [5]. It is proven that for any graph with more than 2 nodes, there will exist a permutation of either STSP or ASTP where NN results in the worst tour [5].

Another limitation the NN algorithm face is that it tends to not yield the optimal solution. As such, a variation of NN – Repeated NN (RNN) was introduced to mitigate this problem. In RNN, we perform the NN algorithm starting at every node of the graph and the path with the optimal weight will be chosen. Since RNN runs the NN algorithm over *n* number of starting nodes, the time complexity of RNN would be $\Theta(n^3)$. RNN however still suffers from the poor performance in STSP and ATSP, and occasionally only yields a path that is only better than at most n-2 possible paths, where n ≥ 4 [5], indicating a serious under-performance in finding optimal paths still.

Question 2 (b)

In this section, graphs will be presented pictorially instead of using an adjacency matrix to provide better readability. Assume that the NN algorithm starts at Node A.

Best Case	Sub-Optimal Case
	
In this example the path computer by the NN algorithm would be optimal. The path would be: **A -> G -> D -> F -> E ->B -> C -> A** This results in a total cost of 8, and is the lowest possible cost for this graph.	In this sub-optimal case, the path computed by the NN algorithm would be: **A -> G -> D -> F -> E ->B -> C -> A** This results in the highest possible cost of 2006. A possible optimal path would instead be: **A -> G -> C -> D -> F -> E -> B -> A** This will result in the lowest cost of 10.

Question 3

Question 3 (a)

The best case for the hybrid sorting algorithm outlined in the question would be an input array of already sorted numbers.

It is assumed that the insertion sort and merge sort algorithm is the exact same as the lecture. As such, the algorithm will not be modified to perform more intelligent tasks that may reduce search time (i.e. always check if the last element in the first sorted subarray is smaller than the first element in the next sorted subarray to be merged. If yes then we do not need to perform any further key comparisons and can merge).

To begin, we pass in the best case scenario of an already sorted array of numbers of length $n = 2^k$ for some integer $k > 3$.

The input is then divided recursively until there are eight elements in each subarray. There are no key comparisons done when dividing the array into subarrays of eight. When there are eight elements in each subarray, we perform insertion sort. As the subarrays are already sorted, 7 key comparisons will be performed on each subarray of eight elements.

After which, we will recursively perform merge sort to merge each sorted subarray. As we are analyzing the best case scenario, the merging of each **pair** of subarrays of length ℓ would take ℓ number of key comparisons. We can break down the number of comparisons into the insertion sort phase, and the merge sort phase.

In the insertion sort phase, we will have 7 key comparisons for every subarray of size eight. Given that the input size is 2^k, we will have 2^{k-3} number of subarrays of size eight. Therefore in the insertion sort phase, we will perform a total number of key comparison:

$$\mathbf{2^{k-3} \times 7}$$

In the merge sort phase, we will have $\frac{2^{k-h}}{2} = 2^{k-(h+1)}$ pairs to merge at height h where k – 1 ≥ h ≥ 3 (as there should be no “pairs” at h = k, and because the threshold value is $2^3 = 8$). The length of any subarray at height h would be 2^h. As such, in the merge sort phase we will have $\mathbf{2^{k-(h+1)} \times 2^h}$ key comparisons at height h. Therefore, in the merge sort phase we will perform a total number of key comparisons:

$$\begin{aligned}&(\mathbf{2^{k-4} \times 2^3}) + (\mathbf{2^{k-5} \times 2^4}) + (\mathbf{2^{k-6} \times 2^5}) + \cdots + (\mathbf{2^{k-(k-1)} \times 2^{k-2}}) + (\mathbf{2^{k-k} \times 2^{k-1}})\\ &= \mathbf{2^{k-1} + 2^{k-1} + 2^{k-1} + 2^{k-1} + 2^{k-1} + \cdots + 2^{k-1}}\\ &= (\boldsymbol{k} - \mathbf{3})(\mathbf{2^{k-1}})\end{aligned}$$

Finally, putting together the insertion sort and merge sort phases, we get the total number of key comparisons in the best case to be:

$$(\mathbf{7 \times 2^{k-3}}) + (\boldsymbol{k} - \mathbf{3})(\mathbf{2^{k-1}})$$

Question 3 (b)

We are given an input array *B* that is sorted in ascending order, and contains 2^k-1 distinct elements. The height of a heap would be k – 1. We know that there will be 2^d nodes at each depth *d* (i.e. 2^0 nodes at depth 0, implying 1 root node at depth 0). Furthermore, each node at height *h* will have *2h* key comparisons done, as we are essentially creating a min-heap and changing it to a max-heap.

With this information, we can formulate that the total number of key comparisons at each height would be $2 \times height \times number\ of\ nodes\ at\ depth$.

As such, the total number of key comparisons for an input of $2^k - 1$ would be:

$$2(k-1)(2^0) + 2(k-2)(2^1) + \cdots + 2\big(k-(k-1)\big)\big(2^{k-2}\big) + 2(k-k)\big(2^{k-1}\big)$$
$$= 2^1(k-1) + 2^2(k-2) + \cdots + 2^{k-1}(k-(k-1)) + 2^k(k-k)$$
$$= \sum_{n=1}^{k} 2^n(k-n)$$

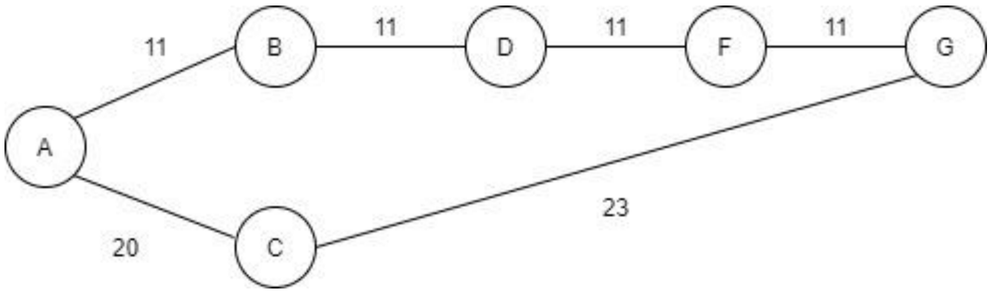
Question 4

The shortest path computed in graph *G'* may not be the shortest path in graph *G*.

Let us assume there exists an optimal path P_1 with n_1 number of edges and a total cost of c_1 in graph *G*. Next, we assume there exists a suboptimal path P_2 with n_2 number of edges and a total cost of c_2 in graph G, where $n_2 > n_1$ and $c_2 > c_1$.

After reducing the weight of every single edge in graph G by 4, we create graph *G'*. In graph *G'*, it is possible for a scenario where $c_1 - (n_1 \times 4) > c_2 - (n_2 \times 4)$, thus making P_2 the optimal path in graph *G'*.

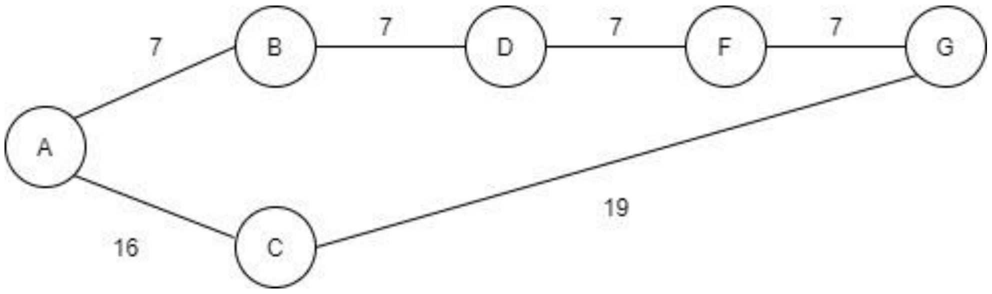
An example would be as follows. Assume a graph *G* that satisfies the constraint stated in the question. Our goal is to find the shortest path from node A to node *G*.



Graph G

As such, the shortest path would be **A->C->G** and the total cost would be **43**. The other sub-optimal path would have cost **44**.

Next, we create a graph *G'* that reduces the weight of every single edge in graph *G* by 4.



Graph G'

In Graph *G'* the shortest path will now be **A->B->D->F->G** and the total cost would be **28**. The previous shortest path of **A->C->G** would no longer be the shortest path as its cost is now **35**.

Therefore, the shortest path from source node *s* computed in graph *G* may not be the shortest path computed in graph *G'*.

References

[1] “Graph Isomorphism,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Graph_isomorphism. [Accessed November 15 2020].

[2] L. Babai, “Graph Isomorphism in Quasipolynomial Time,” *arXiv,* vol. arXiv:1512.03547v2 [cs.DS], 2016.

[3] “Exponential Time Hypothesis,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Exponential_time_hypothesis. [Accessed November 16, 2020].

[4] M. Al-Turkistany (2015, Dec 20). What evidence is there that Graph Isomorphism is not in P? [Blog]. Available: https://cstheory.stackexchange.com/questions/32160/what-evidence-is-there-that-graph-isomorphism-is-not-in-p [Accessed November 16, 2020].

[5] A. Y. A. Z. Gregory Gutin, “Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP,” *Discrete Applied Mathematics,* vol. 117, no. 1-3, pp. 81-86, 2002.