The report should contain a **cover page**, specifying student's full **name:Tiviatis**,

matric number:-,

email:-, and the

tutorial group:DSAI2.

Q1) Yes, they are same topologically, assuming that the name of the vertexes does not matter, graph is undirected, and the edges are all of equivalent weight, they are isomorphic. Bijective mapping of adjacent vertexes preserving structure of graph can be done. For instance, where 1 refers to the left graph (Figure 1) and 2 refers to the graph at the right, let [a1=d2], [b1=b2], [c1=c2], [d1=a2] (Table 1). Each vertex is thus connected to same vertexes as their equivalent.

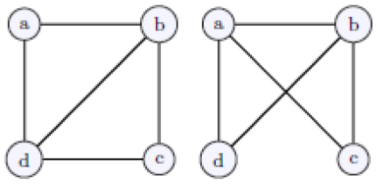

Figure 2:Graphs given in Question

Graph 1	Graph 1 converted	Graph 2	Same?
a1: [b1,d1]	d2: [b2,a2]	d2: [a2,b2]	Yes
b1: [a1,c1,d1]	b2: [d2,c2,a2]	b2: [a2,c2,d2]	Yes
c1: [b1,d1]	c2: [b2,a2]	c2: [a2,b2]	Yes
d1: [a1,b1,c1]	a2: [d2,b2,c2]	a2: [b2,c2,d2]	Yes

Table 1: Working to show graph isomorphism

Intuitively, one should check different permutations of vertex conversion (what I did above is just a single success case) and stop the programme when there is one equivalence found where each node has edges going to the same nodes as the corresponding graph(graphs are isomorphic) or when all possible permutations are exhausted with no equivalence (graphs are not the same). Algorithm in class P have at most polynomial worst-case running time, and algorithm in NP allows verification of possible solutions within polynomial time. The solution cannot be 'found' by algorithm, instead we are attempting to find all possible permutations (amount of cases is factorial: 4! for graph with 4 nodes) of nodes in 2nd graph with respect to the 1st, where 1st graph remains a,b,c,d. Using the cases, we can verify if it's a potential solution(verifies neighbour nodes), which can be done within polynomial time, making it NP.

Theoretically, this question is a graph isomorphism question. The best current known solution (not yet widely accepted) is by László Babai algorithm which aims to reduce the problem instance to quasipolynomially-many smaller, constant instances until such recursive reductions can be resolved with brute force, leading to an overall quasipolynomial-time algorithm[1] of $2^{O((\log n)^3)}$, which makes it an NP problem being neither P as it's not yet within polynomial, nor NP-complete (assuming P≠NP) since if a problem in NP is NP-complete, every other problem in NP is reducible to that problem in polynomial time. Since NP-hard is at least as hard as NP-complete, (plus its solution need not be verifiable in polynomial time), theoretically this is not a NP-hard problem.

Q2a) The algorithm I want to propose is commonly known as the Christofides Algorithm with 1.5-approximation (this means algorithm gives a solution trapped within a factor of 1.5 of the optimal solution). The assumptions for this algorithm are: **(a)** distances of graph instances form a metric space (i.e. fulfils triangle inequality); Assuming P≠NP, if graph is non-metric, there is no α-approximation where α≥1. **(b)**To work graph should be symmetric (weight of edges should be same bidirectionally; undirected graph) since Prim's algorithm is used. **(c)**Step (II) requires initial edge weight to be positive for minimum-cost perfect matching. **(d)**Step (V) requires graph to be Eulerian (contain 0 or 2 odd degree vertexes). **(e)**Only Hamiltonian-cycle considered such that no repeating nodes is travelled. This algorithm can be split into several steps:

I) <u>Form Prim's minimum spanning tree(MST) from graph:</u> Initialize first vertex randomly into tree, select edge with least weight, add into tree and connect new vertex, continue previous steps until no more vertex left. (Possible Improvements: Fibonacci heaps for priority queue to reduce time complexity)

II) <u>With odd degree vertexes, construct a subgraph via minimum cost(weight) perfect matching:</u> First find all vertexes with odd amount of edges using %2; put into matrix. Run

[1] Babai, L. (2019). Canonical form for graphs in quasipolynomial time: Preliminary report. Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing - STOC 2019, 1237-1246. doi:10.1145/3313276.3316356

networkx.algorithms.matching.max_weight_matching[2], which implements a modified "Edmond's Blossom Shrinking Algorithm", but it finds the maximum cost instead, therefore multiply all edges by negative first before starting matching(min. now reversed to max.). Graph with minimum cost perfect matching(a.k.a. sets of odd degree nodes are now connected to each other through minimally weighted edge) should be output. (Possible Improvements: Vladimir Kolmogorov's Blossom V[3] as minimum cost perfect matching algorithm).

III) Combine (I) and (II) graphs: networkx.MultiGraph() allows storage of multi-edges for undirected graphs. If edges connect to vertexes not present, corresponding vertex will form. If edge already exists, additional edge formed with specific identifier via key value.

IV) Euler tour graph from (III): Hierholzer's algorithm is used here to find Euler cycle. 2 stacks are made to store temporary path and final Euler tour path respectively. Starting vertex should have odd degree unless graph contains no odd degree vertices then a vertex is randomly chosen. Starting vertex is placed into first stack. It pushes vertex connected to a random edge into 1st stack and removes that edge from the graph. If all paths of the vertex at top of first stack is exhausted, vertex is popped and pushed into 2nd stack. Process continues until there is no more edges in graph. 2nd stack contains Euler tour path.

V) Delete duplicating vertexes from (IV) and make a Hamiltonian circuit: Each vertex has stored boolean value in another list, iterate through Euler path and update boolean from F to T. If already T, remove vertex (already exist in path). Output is optimal path.

Time complexity: V is number of vertexes and E is number of edges. **(I)** takes O(VlogV + ElogV) = O(ElogV) incurred since all vertexes are inserted into priority queue which spans for logarithmic time. Since in TSP, all vertexes are generally connected, thus $E=V(V-1) \approx V^2$ if V is large and graph is complete, thus $O(E\log V)=O(V^2\log V)$. **(II)** networkx.algorithms.matching.max_weight_matching has time complexity[4] of $O(V^3)$ and the iteration to obtain odd degree vertexes requires V comparisons. The iteration to make all edges negative to use max_weight_matching has time complexity bounded within E (since only vertexes with odd degree are considered and their edges are stored in matrix). Thus time complexity for this step is $O(V^3+V+E)=O(V^3+V+V^2)= O(V^3)$. **(IV)** Each edge in graph is traversed only once and removing an edge takes O(1) time, therefore complexity is $O(E)=O(V^2)$. **(V)** Iterating through Euler path takes $O(E)=O(V^2)$. In conclusion, since all time-complexities of all parts are added for final, with **(II)** having the worst complexity of $O(V^3)$, total complexity of Christofides Algorithm is within $O(V^3)$.

Q2b) Optimal solution obtainable by graph α (Figure 2), and suboptimal solution obtained by graph β (Figure 4), their optimal paths are drawn in dark blue. There are 2 optimal paths in graph α: abcdefa/bacdefb(weight 8.5) and both are obtainable through algorithm in (2a) no matter which node is randomly chosen by Prim's algorithm during creation of MST. For graph β possible suboptimal paths obtained by (2a) are abcdefga(weight 18) and dabcgfed(weight17.5) (optimal is acdefgba with weight 16). Figures 3 and 5 shows the paths taken by Christofides, where the Prim's minimum spanning tree is drawn in light blue and subgraph made by minimum cost perfect matching is drawn in green. Combining both happen to give a Hamiltonian result (i.e. Euler tour of combined graphs contains no repeats).

[2] Networkx.algorithms.matching.max_weight_matching. (2020, August 22). Retrieved November 10, 2020, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.matching.max_weight_matching.html

[3] Kolmogorov, V. (2009). Blossom V: A new implementation of a minimum cost perfect matching algorithm. Mathematical Programming Computation, 1(1), 43-67. doi:10.1007/s12532-009-0002-8

[4] Networkx.algorithms.matching.max_weight_matching. (2020, August 22). Retrieved November 10, 2020, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.matching.max_weight_matching.html

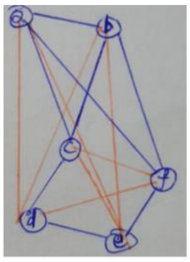
Figure 2: Graph α

	a	b	c	d	e	f
a		1	2.1	3	4	3
b			1.5	3	3	2.5
c				1	1.5	1.5
d					1	2
e						1
f						

Table 2: Weight of edges of graph α

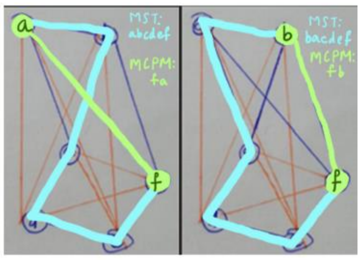

Figure 3: Graph α MST and minimum cost perfect matching subgraph

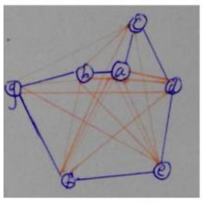
Figure 4: Graph β

	a	b	c	d	e	f	g
a		0.5	1.5	1	4	4.5	4
b			2	3	4.5	4	2.5
c				2	5	5.5	4.5
d					2.5	5	5
e						3	5
f							4
g							

Table 3: Weight of edges of graph β

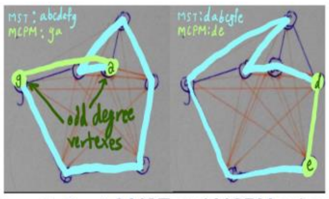

Figure 5: Graph β MST and MCPM subgraph which starts from nodes a and d respectively

Q3a) My interpretation of the question is that insertion sort is used by the mergesort algorithm for mergeSort(n) where n is smaller than 8. Since mergeSort recursively calls itself, it means the smaller blocks of data size 8 and below are all sorted via insertion sort before being sorted via mergesort to combine to form a larger sorted data. Intuitively, an ordered array works best for insertion sort which causes n-1 comparisons for n size data, thus initial data should already be sorted for every group containing items between 8(inclusive) to 4(exclusive since there will be no more division of 2 after 8 is size data), where group size is dependent on size of data (E.g.96 items will give 16 sets of 6 items when continuously divided by 2 and every group of 6 should already be sorted within themselves). Let us call this number trapped between 4 to 8 (2^2 to 2^3) the *factor*. Thus, insertion sort is called via *insertionSort(factor)*. Assuming size of data n = 2^k, for some integer k > 3, comparison*(cmp)* made by insertion sort is $\frac{n(factor-1)}{factor}$. E.g.($\frac{96\ items}{6} = 16\ groups\ of\ 6,$ $16grps \times ((6-1)cmp\ per\ grp) = 80cmp$) Replacing factor by 2^2 or 2^3, cmp done by insertion sort are trapped between: $\frac{2^k(2^3-1)}{2^3} = 2^k - 2^{k-3}\ (inclusive)$ & $\frac{2^k(2^2-1)}{2^2} = 2^k - 2^{k-2}(exclusive)$.

On the other hand, mergesort functions most efficiently when there is only a single key cmp for every 2 items at any given level. From this condition, a possible best case occurs when all the values are the same→if every item is the same, graph is already sorted and it will give best case for insertion sort (1 comparison per item); additionally, it allows the mergesort to place 2 items into new list after each cmp since they are equivalant. This reduces number of cmp since the “larger item” does not have to compare itself with another item from the opposing list. Number of cmp done by each level of a mergesort tree is consistent(e.g. 2nd highest level has 4 sets of 24 items, where each sets takes 24/2=12 cmp, which gives12cmpx4sets=48 cmp in total; highest level has 2 sets of 48 items, each with 48/2=24 cmp, which gives 24cmpx2sets=48 cmp as well.) Thus what we want is the depth of the tree, which from the assumption from the previous part (n = 2^k) gives a depth of k, where there are k levels of comparisons. However, not all levels carry out mergesort: 2-3 levels are forfeited via usage of insertion sort (depending on the factor), thus only (k-2) or (k-3) levels carry out mergesort. Thus comparisons made by mergesort is trapped within $\left(\frac{2^k}{2}\right)\text{x}(k-2) = (k-2)2^{k-1}$ & $\left(\frac{2^k}{2}\right)\text{x}(k-3) = (k-3)2^{k-1}$. Total comparisons made from merged algorithm are therefore trapped between $2^k - 2^{k-3} + (k-3)2^{k-1}(inclusive)$ & $2^k - 2^{k-2} + (k-2)2^{k-1}$, depending on the factor. For a cleaner algorithm, we can assume that number of nodes are 2^k where k is integer. This way, due to multiplicative properties of 2, the only factor attainable is 8. This allows total comparisons to be$\frac{n(8-1)}{8} + \frac{n}{2}(k-3) = \frac{2^k(2^3-1)}{2^3} + \left(\frac{2^k}{2}\right)\text{x}(k-3) = 2^k - 2^{k-3} + (k-3)2^{k-1}$.

Q3b) Floyd's buildheap algorithm than J.W.J. Williams' will be used here. Firstly, a binary heap will be constructed to store items in array (Figure 6). As seen in the figure, there is a

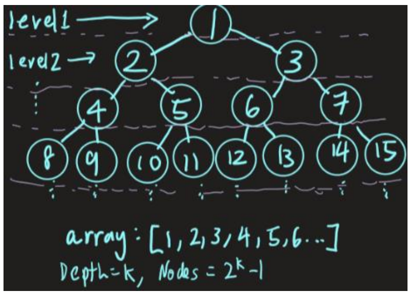

Figure 6: Constructed Binary Heap

depth of k, however to analyse, treat lvl4 as the base level. Since this is made from an increasing array, to allow heap to be a maximum heap, it must be compared from bottom up, right to left(Floyd) to allow nodes to be swapped. First comparison is *15* and *7* and there will be a switch made. Irregardless of switches or values, all nodes at this level (the base level) will trigger a comparison. Now we look at comparisons at lvl3 with lvl2.The array is incremental thus in this entire binary heap, as we move up lvls, the lvl-to-lvl comparisons will results in switches(e.g.after previous comparison,*15* is now at lvl3, and when compared to lvl2, will swap with *3*). When a swap is made, it is important to note that all child nodes below it will also undergo comparison. Since the entirety of lvl2 will be swapped with lvl3 nodes, total comparisons made is the sum of lvl3&4 nodes. Therefore, for every lvl of comparisons, all bottom lvls will also undergo comparisons(e.g. lvl4-3 comparisons causes all lvl4 nodes to compare, lvl3-2cmp causes all lvl4&3 nodes to compare, lvl2-1cmp causes all lvl4&3&2 nodes to compare.) For any lvl j, number of nodes in that lvl is 2^{j-1}. Thus base lvl (lvl k) with 2^{k-1} nodes will be compared k-1 times (instead of k since level 1 does not have any further level to compare itself with) and lvl(k-1) with 2^{k-2} nodes will be compared k-2 times. The pattern continues until lvl 2 with 2 nodes (2^{2-1}=2) will be compared 1 time. Thus total comparison gives a summation of $\sum_{i=1}^{k-1}[(2^{k-i})(k-i)] = \sum_{i=1}^{k-1}(2^i)(i)$. Differentiating $\sum_{i=1}^{k-1}(2^i) = \frac{q(1-q^{k-1})}{1-q}(Geometric\ Progression) \rightarrow$ $\sum_{i=1}^{k-1}(i2^{i-1}) = \frac{d}{dq}\frac{q-q^k}{1-q} = \frac{(1-kq^{k-1})(1-q)}{(1-q)^2} = \frac{1-kq^{k-1}}{1-q}$. Now we multiply both sides by 2 and replace q with 2: $\sum_{i=1}^{k-1}(i2^i) = \frac{2(1-k2^{k-1})}{1-2} = -(2-k2^k) = k2^k - 2$. Thus comparisons taken: $k2^k - 2$.

Q4) Assuming that the distances change as the weight changes but distance changed does not affect position of other nodes with respect to its connections(all nodes still connect to their previously connected nodes), the graph formed post transformation may not be the same. This is a subset of what is known as dynamic weight implementation, and its semi-dynamic since it is decremental. As compared to a linearly transformed graph which will give back same optimum path, in such transformations, after all nodes are altered equally, we may not obtain same optimal path. For instance, the sum of the fastest path from G(Figure 7) is not the lowest after transformation: shortest path from s→f in G (s,f) takes 36→ 32 after transformation but there exist shorter path(s,h,g,f) in G': 8+11+8=27(Figure 8). Nevertheless, take note that if optimal paths post-transformation stays the same(Figures 9 &10), and there is an alternative optimal path, since the edges around a particular node are reduced by the same amount, the greedy algorithm will function in same sequence since it looks at optimal choice(optimal property is proven in slides) at any given step first, thus even if there exist 2 optimal path from vertex a→c, in both A and A', the obtained paths will be the same. Therefore, the optimal paths obtained by Dijkstra's algorithm Greedy search should be consistent per run, and difference in results are due to transformation which changes optimal weight, thus changing optimal paths after a run.

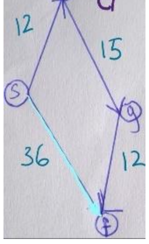

Figure 7: Graph **G**

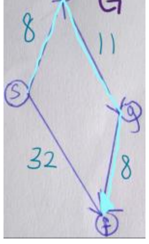

Figure 8: Graph **G'**

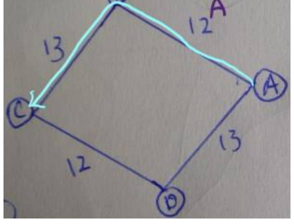

Figure 9: Graph **A**

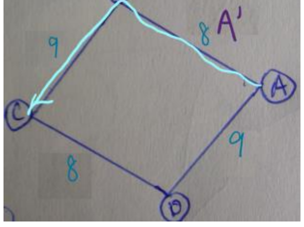

Figure 10: Graph **A'**