

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

CZ2001: Algorithms
Individual Assignment Report

Submitted by: Ng Chi Hui
Matriculation Number: U1922243C
Tutorial: Group 1

School of Computer Science and Engineering

Q1:

In this question we assume that the graph on the right is G and the graph of the left is H.

To better understand whether the problem is NP, NP-hard, NP-complete or P class, we can first try to prove whether the two graphs are isomorphic. This problem is a decision problem of True or False. It is often easier to prove that the two graphs are not isomorphic thus we can try to check these conditions first. First, they have the same number of vertices: 4. Second, they have the same number of edges, 5. Third, they have the same degree sequence is {2, 2, 3, 3}. This shows that the two graphs might be isomorphic (Prolubnikov, n.d.).

We then try to construct an isomorphism, we are able to show that $f(d) = a$, $f(b) = b$ and $f(c) = c$ and $f(a) = d$. This defines a bijection from the vertices of G to the vertices of H. We can then check that the adjacent vertices of G are mapped to the adjacent vertices of H by constructing their adjacency matrices (fsu, n.d.).

The adjacency matrix for G (vertices are listed in the order of d, b, c, a)	The adjacency matrix for H (vertices are listed in order of a, b, c, d)
$A(G) = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$	$A(H) = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$

Since $A(G) = A(H)$, adjacency is preserved under this bijection. Hence the graphs are isomorphic.

Moving on, we can now focus on discussing what kind of class of problem is graph isomorphism in. We can observe from the above example the two graphs are only isomorphic iff both the adjacency matrices of both graphs are the same. Using the adjacency matrix method as described above, we can come up with a generic algorithm. The algorithm says first we can save all the vertices of G from 1 to n as adjacency matrix and do the same for graph H. Then we can compare the adjacency matrices of G and H if they are identical they are isomorphic. If not we go back to reconstruct the matrices again for graph G and H using a different set of nodes. If all the possible combinations have been found for graph H and there's no match the two graphs are not isomorphic. In the actual implementation this can be done using list or hash maps.

Thus, in the worst case scenario where the two graphs are not isomorphic and by using brute force search we will be checking all possible solutions, the total number of bijections/comparisons done will be $n!$ between the vertices of two n vertex graphs. The computer will have to check $n!$ combinations of adjacency matrices for H. This means that the problem cannot be solved in polynomial time. As for the verification, it will take $O(n^2)$ time to check whether some bijection of $V(G)$ onto $V(H)$ is isomorphic. Therefore, the graph isomorphism problem belongs to the NP class of problems. Graph isomorphism is an element of NP class because its set of solutions can be verified in polynomial time but it cannot be solved in polynomial time (Baeza, 2016).

This problem can be classified as an NP problem instead of NP complete and NP hard as currently there are no known way to reduce the issue of graph isomorphism.

Q2a)

The travelling salesman problem is a NP problem. Using the brute force method, we will arrive at $(|v|-1)!/2$ permutations giving a time complexity of $O(n!)$. To solve the TSP algorithm, the Nearest Neighbours algorithm can be used.

The nearest neighbour algorithm first marks all the cities (vertices) as unvisited, "u" it then chooses an arbitrary city as its starting point and mark it as current vertex, "v". At each step, it chooses the nearest unvisited city, "u" connected to the current city, "v" then marked as visited and set as current vertex. This step is being repeated until all the cities have been visited. The last step will be to return to the starting city (mtu, n.d.).

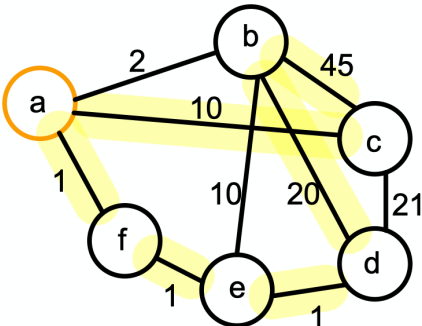
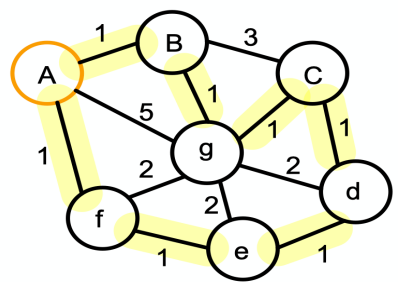
Time complexity

By searching edges for every city that has not yet been visited requires at most $O(v)$ operations, which is the number of times the while loop will be executed. With each city only being considered only once the total time complexity for nearest neighbour algorithms will be $O(v^2)$.

Limitation

The nearest neighbour algorithm is a greedy algorithm meaning the it chooses to make the most optimal choice where it picks the edge with the least weight at each step as it seeks to solve the entire problem. However, it will not always give us the most optimal solution. This is because sometimes, the algorithm will have to add an edge that has a very high weight because all the other edge that have lighter weights and connected to this node have already been visited. If a poorer choice (i.e suboptimal edge) was picked first, it might have led to better results in the end. However, as a greedy algorithm, it is very short sighted where it doesn't consider all the options available and it will always makes the most optimal choice at that point in time overall might have led to a worst results.

2b) We will start from Node A and end at node A.

Not-the best solution	Best Solution
 <p>Using the nearest neighbour algorithm, the path generated will be $a \rightarrow f \rightarrow e \rightarrow d \rightarrow b \rightarrow c \rightarrow a$. The total distance taken will be 78. However, just at a glance we are able to tell that one of the more optimal paths will be from $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow a$. The total distance taken will be 71 instead.</p>	 <p>Using the nearest neighbour algorithm, the best route generated from a will be $a \rightarrow f \rightarrow e \rightarrow d \rightarrow c \rightarrow g \rightarrow b \rightarrow a$. The total distance will be 7. This will be the same case if a brute force algorithm is being used. Thus this is the best solution</p>

3a)

In this hybrid algorithm, insertion sort occurs when each of the original array has been divided into each size of 8 elements in the array during merge sort. Thus, the number of arrays that will be sorted with insertion sort will be $2^k/8 = 2^{k-3}$. Since the array is already sorted during insertion sort, in the best case, each time a new element is inserted from the unsorted array, it only needs to compare with its left neighbour in the sorted array, with its left neighbour already smaller than it, no swaps are being carried out. Overall, $n - 1 = 8 - 1 = 7$ key comparisons are executed in the best case of **one** array sorted with insertion sort. In this hybrid algorithm, the number of arrays that will be sorted with insertion sort will be 2^{k-3} arrays each of size 8. Thus the total number of insertion sort comparisons will be $7 * 2^{k-3}$ comparisons done at this stage.

The best cases of merge sort occurs when the input already has already be sorted. This means that for every pair of 2 sub lists, when both the lists are sorted, the all the elements in the right sub lists will be smaller than the first element of the left sub lists. During merging, every element in the left array can be compared to the only the smallest element in the left array and put into the sorted subarray then the sorted array on the right will also not need any further comparisons and can be moved into the same new sorted subarray. Thus, merging will always take $n/2 = 2^{k-1}$ comparisons as only one element from the right subarray will be compared. For the initial size of input array 2^k the number of comparisons made will be 2^{k-1} and the total number of merge operations taken will be $(k-3)$. Therefore, the total number comparisons for the entire array during merge will be $(k-3) (2^{k-1})$

The total number of key comparisons for the hybrid sort will be the sum of comparisons from the merge sort and insertion sort which equals to $(k-3)(2^{k-1}) + 7 * 2^{k-3}$. We must give the answer in terms of n , thus we have $(\log_2 n - 3)(n/2) + 7/8n$

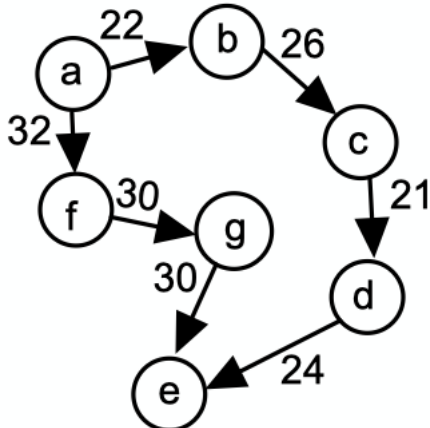
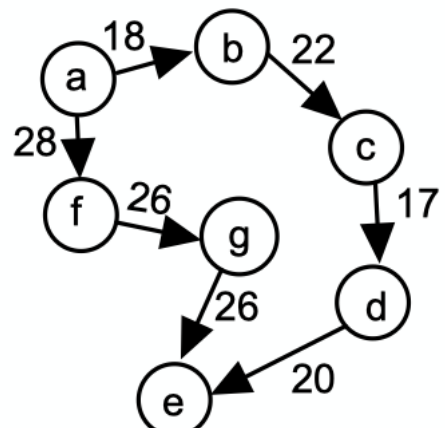
3b)

K	Total Number of Comparisons	Workings
1	0	$= 0$
2	2	$= 2^1$
3	8	$= 2^2 + 2(2^1)$
4	22	$= 2^3 + 2(2^2) + 3(2^1)$
5	52	$= 2^4 + 2(2^3) + 3(2^2) + 4(2^1)$
:	:	:
k-1		$= 2^{k-2} + 2(2^{k-3}) + 3(2^{k-4}) + 4(2^{k-5}) + \dots + k(2^1)$ (arithmetico-geometric progression) $= \sum_{j=1}^{k-1} j(2^{k-j}) = 2(2^k - k - 1)$

By drawing out the construct heap method, we are able to find out the number of comparisons given the total number of models. We can then model after the number of comparisons to come up with equations to solve it. K stops at $k-1$ as there is no comparisons done at the last level. The main comparisons occur in the fix heap function where it makes 2 comparisons swaps the root with the larger value in the child node.

Q4

No, the shortest computed path will not necessarily be the same. Suppose, for example that we will like to find the shortest path from a to e.

Graph G	Graph G'
	
<p>Using the Dijkstra algorithm, the shortest computed path will be from $a \rightarrow f \rightarrow g \rightarrow e$ with total weight 92.</p> <p>Whereas the other path $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ has a total weight of 93 will not be chosen.</p>	<p>After reduction the shortest path from a to e will be $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ with a total weight of 77.</p> <p>The initial shorter path of $a \rightarrow f \rightarrow g \rightarrow e$, now has a higher total weight of 80 instead.</p>

These cases often happen, when there's two path A and B which have very similar total weight before the reduction and but differs widely in the number of edges they have. Say path A has total weight of 100 with 20 edges and path B has total weight of 98 with 2 edges. Thus after reducing by 4 for each edge. The total weight of path A = $100 - 20(4) = 20$ whereas the total weight for path B will now be heavier $98 - 2(4) = 90$.

Bibliography

Prolubnikov, A. (n.d.). Retrieved from <http://ceur-ws.org/Vol-1623/paperco14.pdf>
fsu. (n.d.). Retrieved from https://www.cs.fsu.edu/~lacher/courses/MAD3105/lectures/s2_1graphintro.pdf
Baeza, L. (2016, April 24). Retrieved from https://www.projectrhea.org/rhea/index.php/Walther_MA279_Spring2016_topic9
mtu. (n.d.). Retrieved from <http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2028%20-%20Approximation%20Algorithm.htm>
CZ2001 Lecture Notes