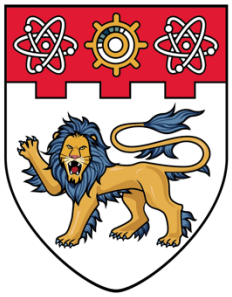


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ2001 Algorithms Individual Assignment

Yong Wen Shiuan
U1922037H
C190170@e.ntu.edu.sg
Tutorial group: SS3

1) Technically, the 2 graphs are not exactly the same, because some edges, e.g. edge (a, c), exist in the 2nd graph but not in the first graph. In this example, if node a in the 2nd graph was simply relabelled as node d and vice versa, both graphs would be exactly the same. Most properties of a graph do not depend on the particular labelling/names of the vertices, as they can be simply relabelled based on a mapping to make them exactly the same. In particular, a common notion of graph similarity is graph isomorphism (**GI**), where 2 graphs G and H are isomorphic if there exists a mapping / (edge-preserving) bijection of vertices in G to vertices in H, i.e. $x \rightarrow f(x)$, where $x \in V(G)$ and $f(x) \in V(H)$, where edge $(x,y) \in G$ if and only if edge $(f(x),f(y)) \in H$. For this example, if we map from graph 1 to graph 2: $a \rightarrow d$, $b \rightarrow b$, $c \rightarrow c$, $d \rightarrow a$, the GI property is fulfilled. Hence, if the exact labelling of the vertices doesn't matter, we can take isomorphic graphs (e.g. the 2 graphs pictured) to be the same graphs that convey the same information. But what is the computational complexity of GI?

Is GI a P class problem? I.e., is there an algorithm that runs in polynomial time and can determine if 2 graphs are isomorphic? While there exists a polynomial time solution for some special cases such as Planar Graphs (e.g. those 2 graphs in question 1) and trees, I assume that the question is referring to GI for any graph, and not planar graphs specifically. Currently, determining if **any** 2 graphs are isomorphic is not known to be solvable in polynomial time, hence it is not known for sure if $GI \in P$. As of now, it seems that currently the fastest confirmed algorithm for solving the GI problem runs in exponential time, although if a deterministic polynomial time algorithm that solves GI is discovered in the future, this would mean $GI \in P$.

Is $GI \in NP$? I.e. given some certificate(s) (a proof that the answer is Yes to the decision problem, e.g. an edge-preserving bijective mapping in this case, somehow generated in polynomial time), does there exist a verification algorithm that runs in polynomial time to verify the certificate(s), i.e. if the mapping achieves the GI property? Given 2 graph adjacency matrix representations G and H, one can first check if they have same number of vertices. If true, one can somehow (maybe by a lucky guess) get a certificate / edge-preserving mapping, which maps each vertex x, where $x \in G$, to a unique vertex f(x), where $f(x) \in H$. The mapping of each vertex can be done in O(|V|) polynomial time. Using that mapping, one can check for each pair of vertices $x, y \in G$, that edge $(x, y) \in G$, if and only if edge $(f(x), f(y)) \in H$. We can use two nested loops, each iterating |V| times, to iterate through every pair of vertices (x, y) in both graphs. For each pair, we compare to see if G(x,y) == H(f(x), f(y)) based on adjacency matrix values. We accept iff every pair satisfies the condition above, i.e. (x,y) == H(f(x), f(y)). This verification can be done in polynomial time $O(|V|^2)$ (2 nested loops). Hence, given a nondeterministic program that somehow generates the 'right' guess, if any exist, mapping can be done in polynomial time and it can also be verified in polynomial time. Therefore $GI \in NP$.

After establishing that $GI \in NP$, one may ask, is GI NP-hard? I.e. is GI at least as hard as all NP problems? If $GI \in$ NP-hard, this would mean GI is NP-complete (since it would be both NP and NP-hard). If one can show how to reduce any known NP-complete problem to the GI problem, then $GI \in$ NP-complete. So far, no one has managed to do so, so it is not known for sure if $GI \in$ NP-complete. Although there is some evidence which indicates GI is unlikely to be NP-complete: For one, the counting version of GI is polynomially reducible to its decisional version; a property not observed in the counting version for all known NP-complete problems. Also, GNI (graph non-isomorphism) $\in$ class AM (Arthur-Merlin) implies that if GI were NP-complete, the polynomial time hierarchy would collapse to its second level, something that's unlikely.

In summary, $GI \in NP$, but GI is one of the few standard problems where it is currently not known to belong in either of its (if $P \neq NP$, disjoint) subsets: P and NP-complete.

2a) Nearest neighbour (NN) approximation algorithm. Its time complexity shall be analysed in terms of no. of comparisons needed to get the minimum edge weight at each iteration/ node. Assuming an undirected weighted graph is used, (arbitrarily) select a starting source vertex s, to start the cycle C; and assign s to v. While there are vertices not in C, select the 'nearest neighbour', or edge (v, w) with minimum weight and vertex w is not in C. If there is more than 1 possible edge (v, w), to find the edge with minimum weight, one has to compare the values of all possible edges (v, w). Then assign v as w. So this is done at every iteration until all vertices are in cycleC, and then add the final edge (v, s). **Assuming the code initialises minimum_distance as Integer.MAX**, we can express maximum number of comparisons (e.g. in a complete graph) for all iteration as such:

Time complexity = (|V| - 1) + (|V| - 2) + (|V| - 3) + … + (|V| - (|V|-1)) //each + (…) + is 1 iteration

$= \sum_{i=1}^{|V|-1}(i)$ //an arithmetic sequence

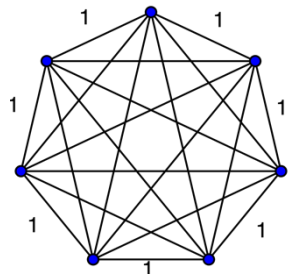
$= (|V|)(\frac{|V|-1}{2}) = 0.5|V|^2 - 0.5|V|$

Hence time complexity of NN approximation algorithm = O ($|V|^2$). It will always be ($|V|^2$) time taken(i.e. in best/worst/average case) if a complete graph is used, which is usually the case. But the time taken may be reduced slightly if the nodes are not of maximum degree and if an adjacency list is used instead of an adjacency matrix.

Limitations: 1 limitation is that it will make the greedy choice at every step, but to complete the cycle, it has no choice but to traverse the edge from last visited vertex to the source vertex, no matter the cost, which potentially affects optimality of the solution (e.g. in part b)). In fact, NN produces the worst route on some specially crafted graph(s) (Gutin et al., 2002).
-Another limitation of NN, is that different routes may be returned if it starts at different cities. For the same graph, the route returned if starting at 1 city gives the optimal solution, but the route returned if starting at a different city gives a non-optimal and costlier solution (e.g. in 2b)). Also, depending on how tie-breaking of more than 1 equal minimum edge-weights is done, the route returned may change from optimal to non-optimal (also shown in 2b)). This could be resolved by performing NN on every vertex (which would make running time O ($|V|^3$).), or adding additional heuristics to inform which is the ideal vertex to start from and how to break ties optimally.

2b) assuming non-euclidean/non-metric TSP; i.e. asymmetric TSP, where triangle equality does not hold, and using a complete weighted undirected graph with 7 nodes:

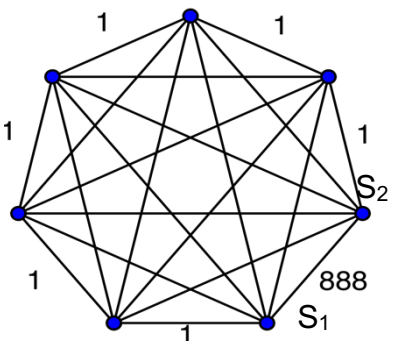
Graph G_{opt}:



Optimal solution: Graph G_{opt}, as seen on the right, where only the edges at the edges of the graph, → , have weight 1; **all other edges (i.e. the ones in the middle) have weight 2**. Using NN, starting from any (arbitrary) vertex v ∈ V(G_{opt}), we will always arrive at the optimal solution, the route of length 7 (1 x 7 = 7).

Non-optimal solution: Graph G_{non_opt}, as seen on the right, which is essentially the same as Graph G_{opt}, just that **Edge (S_1, S_2) has weight 888**. Like graph G_{opt}, the unlabelled edges of graph G_{non_opt}, have weight 2. Non-optimal solution will definitely be returned if NN begins at either S_1 or S_2. Resultant route returned will be of cost 894 (888 + 1 x 6). However, the optimal solution is of cost 9 (1 x 5 + 2 x 2), which may be obtained if one starts the NN from any other vertex v ∈ V(G_{non_opt}), **but is not S_1 or S_2**. Non-optimal solution may also be returned if its starts from any other vertex that is not S_1 or S_2, depending on which edge is selected.

Graph G_{non_opt}:



3a) Assuming the array input size is of size n where $n = 2^k$, where $k > 3$ and k is an integer:
The best case would be if the input array is already sorted, i.e. in ascending order. The procedure is as follows:
First split the array into half, and then check if the resultant sub-arrays are of size 8. If not, continue splitting each resultant sub-arrays in half, and this process is done recursively, until there are only sub-arrays of size 8 left. When the sub-arrays are of size 8, insertion sort is performed on each of them.
For the best case number of comparisons for insertion sort, where the array of size 8 is already in sorted order, there will be 7 comparisons. There are a total of n-1 outer iterations for insertion sort, and with 1 comparison per outer iteration in the best case, B(8) = 8 – 1 = 7.
For mergesort, all the 'key' comparisons occur in the merge portion. At each merge, the best case number of key comparisons also occurs when the array is already sorted in ascending order. In the best case, each element in the left array is compared one-by-one, only to the first element in the right array, to check if it is smaller, before merging the 2 sub arrays together.

Hence, the number of comparisons at each merge to produce an array with n elements, in the best case = n/2.
We can express the number of key comparisons performed by this hybrid sorting algorithm in the best case when running on an input array of size n, B(n), in an equation as follows:
B(8) = 7

$B(n) = B(\frac{n}{2}) + B(\frac{n}{2}) + \frac{n}{2}$

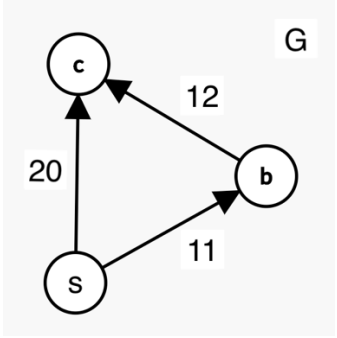
$B(2^k) = 2B(2^{k-1}) + 2^{k-1}$ // since $n = 2^k$
$= 2(2B(2^{k-2}) + 2^{k-2}) + 2^{k-1}$
$= (2^2)\, B(2^{k-2}) + 2(2^{k-1})$
$= (2^2)\, (2B(2^{k-3}) + 2^{k-3}) + 2(2^{k-1})$
$= (2^3)\, B(2^{k-3}) + 3(2^{k-1})$
…
//Form: $(2^m)\, B(2^{k-m}) + m(2^{k-1})$
$= (2^{k-3})\, B(2^{k-(k-3)}) + (k-3)(2^{k-1})$
$= (2^{k-3})\, 7 + (2^2)\, (k-3)(2^{k-3})$ // $B(2^{k-(k-3)}) = B(2^3) = B(8) = 7$
$= (4k - 5)(2^{k-3})$ // $(7 + 4(k-3)) = 4k – 5$
$= \frac{1}{2}\, n \log_2 n - \frac{5}{8}\, n$ //in terms of n, given $k = \log_2 n$ and thus $(2^{k-3}) = \frac{n}{8}$

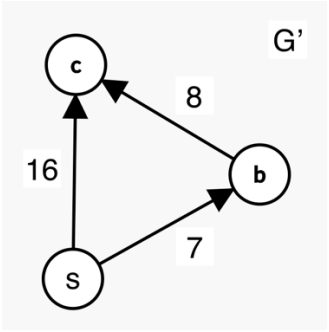
Time complexity in the best case remains Θ (nlogn), same as worst case for merge sort alone.

3b) An input array B contains 2^k-1 distinct elements (k is a positive integer) means B is a full / complete binary tree, since number of nodes $n = 2^k-1$. The worst-case time complexity of heapConstruct(), i.e. W(n), which essentially calls heapifying(), is as follows: fixHeap takes $2\log_2 n$ time whereas heapifying() makes 2 recursive calls to each of its subtrees. Given the worst case when the heap is a single node W(1), no comparisons are made, so we have:
W(1) = 0

$W(n) = W(\frac{n-1}{2}) + W(\frac{n-1}{2}) + 2 \log_2 n$
$= 2W(\frac{n-1}{2}) + 2 \log_2 n$
$= 2(2W(\frac{n-3}{4}) + 2 \log_2 (\frac{n-1}{2})) + 2 \log_2 n$

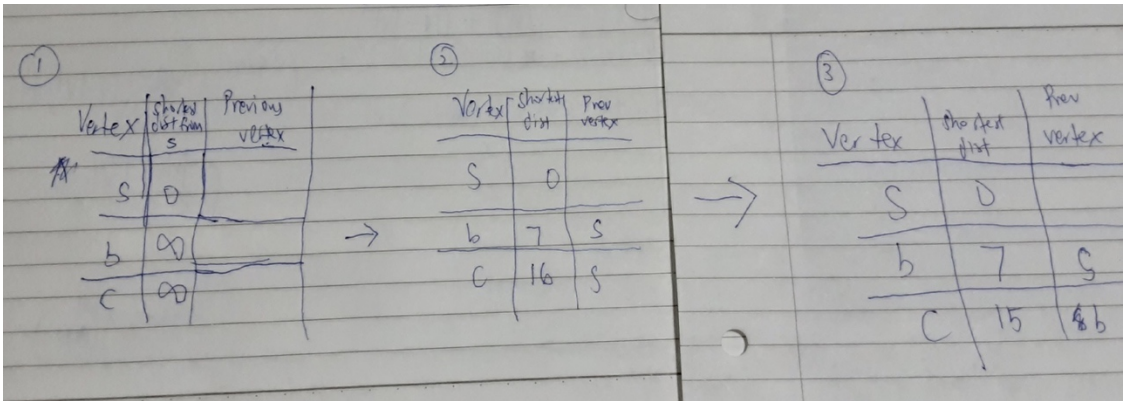
4) No, the shortest path computed in G' is not also a shortest path in G all the time. Consider this counter-example: weighted directed graph G with all edge weights greater than 10:





Reducing the weight of every single edge in G by 4, we get graph G' with all edge weights greater than 6, as seen above on the right.

In G, shortest path from s to c is s → c.
Performing Dijkstra's algorithm on G', we have the following steps: First, we initialise the paths from s to every other vertex to be infinity (1). Next, we initialise the distances to infinity, then we visit all neighbouring nodes of s, so we have (2) as shown below. Next we traverse all untraversed edges from b, resulting in (3) and since all the edges have been traversed, that concludes the algorithm. Resultant shortest path in G', from s to c is s → b → c.



Decreasing 4 from all edge weights of a graph would mean that when calculating shortest paths, alternative paths that use more edges may be favoured, as the total path weight of the alternative path is decremented by 4 x (the number of paths more than other paths), so a shortest path in G may not necessarily be a shortest path in G'.