SC2001/CX2101
Algorithm Design and Analysis

Tutorial 6
Introduction to NP
(Week 13)

This tutorial helps you develop skills in the learning outcome of the course: "Able to classify some decision problems into P or NP problems and apply greedy heuristic approach to solve NP-complete problems".

Q1: Is this problem in the class of P or NP? Justify your answers.

Given a network of cities G and a positive integer k. Are the shortest paths between all pairs of cities not longer than k?

Dijkstra's algorithm is able to compute the shortest path from a single vertex to all other vertices in $O(n^2)$ time.

Running Dijkstra's algorithm from every vertex will find the shortest paths between all pairs of vertices in $O(n^3)$ time.

So checking all the shortest paths can be done in $O(n^3)$ time.

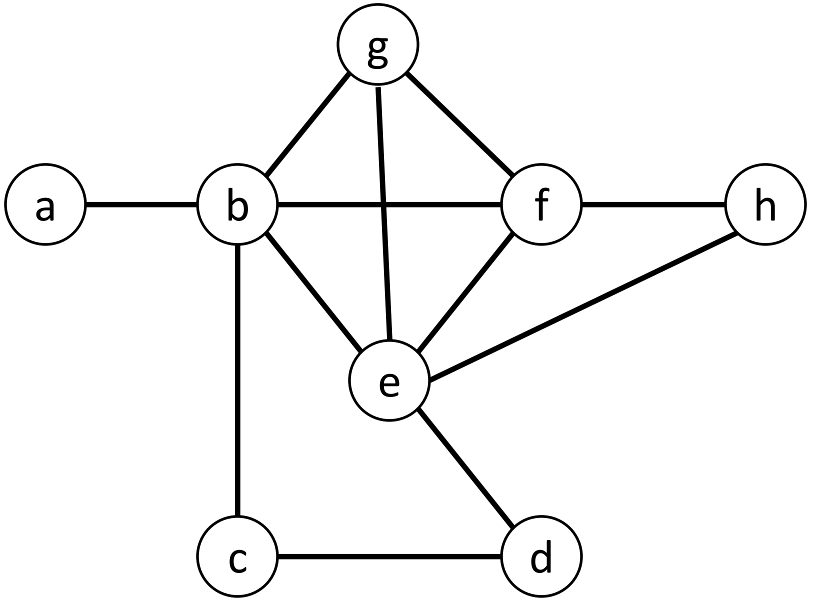
Therefore this is a P problem.

Note the **Floyd-Warshall Algorithm** will compute the all-pairs shortest paths more elegantly in $O(n^3)$ time.

Q2: Show that the clique problem is in NP.

Given a graph $G = (V, E)$, and a positive integer $k \leq |V|$. Does G contain a k-clique? In other words, is there a subset $V' \subseteq V$ such that $|V'| \geq k$ and every two vertices in V' are joined by an edge in E? A clique with k vertices is called k-clique.

E.g. this graph has one 4-clique and a few 3-cliques



Q2: Show that the clique problem is in NP.

Given a graph $G = (V, E)$, and a positive integer $k \leq |V|$. Does G contain a k-clique? In other words, is there a subset $V' \subseteq V$ such that $|V'| \geq k$ and every two vertices in V' are joined by an edge in E?

Prove that clique is in NP:

- Guess a subset V' of V, the verifier accepts the solution if
- Every vertex of V' is in V --- $O(n^2)$
- $|V'|$ >= k --- $O(n)$ or $O(1)$ if V' is given in an array
- For each pair of vertices in V', there is an edge in E that connects them. --- $O(n^2)$ if E is an adjacency matrix
- Thus the solution can be verified in $O(n^2)$ time.

Q3: Show that the 3-CNF-SAT problem is in NP.

The 3-CNF-SAT problem refers to this:

Let $U = \{u_1, u_2, \ldots, u_n\}$ and $C = \{c_1, c_2, \ldots, c_m\}$ where each u_i is a variable and each c_j is a disjunction of 3 variables. The 3-CNF-SAT problem asks if there is a satisfying truth assignment to variables that simultaneous satisfies all the clauses in C.

For example:

$U = \{u_1, u_2, u_3, u_4\}$ and $C = \{\{u_1, \neg u_2, u_3\}, \{\neg u_1, u_2, u_4\}\}$, that is, is there a truth assignment that makes

$(u_1 \vee \neg u_2 \vee u_3) \wedge (\neg u_1 \vee u_2 \vee u_4)$ true?

Q3: Show that the 3-CNF-SAT problem is in NP.

The 3-CNF-SAT problem refers to this:

Let $U = \{u_1, u_2, \ldots, u_n\}$ and $C = \{c_1, c_2, \ldots, c_m\}$ where each u_i is a variable and each c_j is a disjunction of 3 variables. The 3-CNF-SAT problem asks if there is a satisfying truth assignment to variables that simultaneous satisfies all the clauses in C.

Given a problem with n variables and m clauses and a guess of the truth assignment,

1. The verifier evaluates the truth value of one clause in O(1)
2. Evaluating all clauses takes O(m) time.

So a solution can be verified in O(m) time.

Q4: Implementing shortestLinkTSP()

Implement the shortestLinkTSP() algorithm below (slide 29 of lecture notes) to find a TSP tour in graph G. You may consider using a minimizing heap, a union-find data structure and other data structures in your implementation of the algorithm.

1. A minimizing heap pg is used to store the edges of G, the keys of the nodes are the edge weights.
2. A union-find data structure is used to store connected vertices (vertices already in some fragments of TSP tours)
3. An adjacency matrix/list representation of a graph C is used to store the edges chosen for the TSP tour.
4. An array edgeCount to store the number of edges incident on each vertex v in C

shortestLinkTSP(V, E, W)

{ **pq** = minimizing heap of the edges of G;

id = array in the union-find structure, each vertex v in its own component, i.e. id[v] = v;

initialise all elements in **edgeCount** to 0;

C = empty; // C is a graph with no edges

```
    while (no. of edges in C < n - 1)  {
            vw = getMin(pq);
            deleteMin(pq);
            if (not connected(vw.from, vw.to) and
               edgeCount[vw.from] < 2 and edgeCount[vw.to] < 2)  {
                  C[vw.from][vw.to] = 1;
                  no. of edges in C ++;
                  union(vw.from, vw.to);
                  edgeCount[vw.from] ++;
                  edgeCount[vw.to] ++;        }
     }
    add edge connecting the end points whose edgeCounts are 1
to C;
    return C;                  }
```