1. The traveling salesman problem can be defined in two ways:

   - The Traveling Salesman Problem
     - INPUT: A weighted graph G
     - OUTPUT: Which tour $(v_1, v_2, \ldots, v_n)$ minimizes $\sum_{i=1}^{n-1} (d[v_i, v_i + 1]) + d[v_n, v_1]$
   - The Traveling Salesman *Decision* Problem
     - INPUT: A weighted graph G and an integer k
     - OUTPUT: Does there exist and TSP tour with cost $\leq k$

   Suppose we are given an algorithm that can solve the traveling salesman decision problem in (say) linear time. Give an efficient algorithm to find the actual TSP tour by making a polynomial number of calls to this subroutine.

Solution: There are 2 parts to this problem, finding the TSP minimum tour value, then finding the TSP minimum tour.

To find the minimum tour value we use binary search to find the integer value k where the decision problem returns true but $k-1$ returns false.

To find the minimum tour we pick a vertex then remove an edge then recheck the TSP decision problem for k. If the decision returns false then that edge has to be part of the minimum tour. If it returns true then that means the edge was not part of the minimum tour, keep removing edges until you get false. When this edge is found move on to the vertex it connects to. Repeat this process removing previous visited vertices from the neighbor sets.

Note: We keep removing edges in case there are multiple tours with the minimum value. Also we add the last vertex in because once we add the second to last there is only one option and there will be no more unvisited vertices when we check the neighbors of the last vertex.

```
TSP(G(V,E)):
  lower ← |V|*min(E)
  upper ← |V|*max(E)
  k ← BinSearch(lower,upper)
  v ← v_1
  visit ← empty
  while visit ≠ V
      check ← neighbors(v)
      for w in check-visit
          G ← G-e_vw
          if TSPD(G,k) is false and v is not in visit
              G ← G+e_vw
              add v to visit
              v ← w
      if |visit| = |V|-1
          add V-visit to visit
  return visit
```

```
BinSearch(lower, upper)):
  if upper-lower < 1000
      find k via brute force
  else
      mid ← (lower+upper)/2
      if TSPD(G,mid) is true
          k ← BinSearch(lower,mid)
      else
          k ← BinSearch(mid,upper)
  return k
```

The binary search is dependent on edge weights which if they are not too different will be inconsequential. This algorithm loops through the edges and TSPD is linear ($O(V+E)$), so the total running time is $O(E(V+E))$

■

2. A **Hamiltonian cycle** in a graph is a cycle that visits every vertex exactly once. A **Hamiltonian path** in a graph is a path that visits every vertex exactly once, but it need not be a cycle (the last vertex in the path may not be adjacent to the first vertex in the path.)

   Consider the following three problems:

   - *Directed Hamiltonian Cycle* problem: checks whether a Hamiltonian cycle exists in a *directed* graph,
   - *Undirected Hamiltonian Cycle* problem: checks whether a Hamiltonian cycle exists in an *undirected* graph.
   - *Undirected Hamiltonian Path* problem: checks whether a Hamiltonian path exists in an *undirected* graph.

   (a) Give a polynomial time reduction from the *directed* Hamiltonian cycle problem to the *undirected* Hamiltonian cycle problem.

Solution: For any arbitrary directed graph $G_d := \{V_d, E_d\}$, construct the following undirected graph $G_u := \{V_u, E_u\}$:

- $V_u := \{v_{in}, v_{mid}, v_{out} | v \in V_d\}$. For each of the vertices in the directed graph, we split them into a triplet of in, mid, and out.
- $E_u := \{(u_{out}, v_{in}) | (u,v) \in E_d\} \cup \{(v_{in}, v_{mid}), (v_{mid}, v_{out}) | v \in V_d\}$. For each of the triplets that comes from the same vertex, we connect them in the order of in-mid-out. The directed edges in the V_d become the undirected ones that connect out and in between corresponding triplets.

Notice that $|V_u| = 3|V_d|$ and $|E_u| = |E_d| + 2|V_d|$, so this reduction is linear.

$\Rightarrow$: Suppose that in G_d there exists a Hamiltonian cycle $C_d := \left(c_1, c_2, \ldots, c_{|V_d|}\right)$, where $c_i \in V_d$. Then in G_u there should also exist

$$C_u := \left(c_{1in}, c_{1mid}, c_{1out}, c_{2in}, c_{2mid}, c_{2out}, \ldots, c_{|V_d|in}, c_{|V_d|mid}, c_{|V_d|out}\right),$$

which is a Hamiltonian cycle in G_u.

$\Leftarrow$: Suppose that in G_u there exists a Hamiltonian cycle C'_u. By definition, within each of the triplets there should only be a path of order in-mid-out, and between two triplets there should only be an edge of out-in. Thus, C'_u should always be of the following form

$$C'_u := \left(c'_{1in}, c'_{1mid}, c'_{1out}, c'_{2in}, c'_{2mid}, c'_{2out}, \ldots, c'_{|V_d|in}, c'_{|V_d|mid}, c'_{|V_d|out}\right),$$

which corresponds to a Hamiltonian cycle $C'_d := \left(c'_1, c'_2, \ldots, c'_{|V_d|}\right)$ in G_d.

■

(b) Give a polynomial time reduction from the *undirected* Hamiltonian Cycle to *directed* Hamiltonian cycle.

Solution: This reduction is simpler than the previous one. Given an instance G of undirected Hamiltonian cycle, Let G' be the directed graph with the same vertices as G and containing edges $u \to v$ and $v \to u$ for every edge $uv \in G$.

($\Rightarrow$) If C is a cycle in G, then C is also a cycle in the directed graph G'. For every $u \to v \in G'$, the edge uv is in G.

($\Leftarrow$) If C is a cycle in G', then C is also a cycle in the original graph G. For every $uv \in G$, the edge $u \to v \in G'$ by the construction.

■

(c) Give a polynomial-time reduction from undirected Hamiltonian *Path* to undirected Hamiltonian *Cycle*.

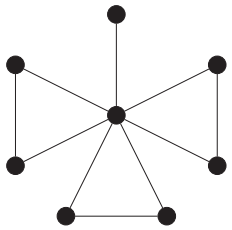
Solution: Let the input to this problem be an undirected graph G. The goal is to produce G' such that G has a Hamiltonian path if G' has a Hamiltonian cycle.

This can be done by adding a vertex v with edges to every vertex in the original graph G, this will be G'.

$\Rightarrow$: If there exists a Hamiltonian path P in G, starting with vertex s and ending with vertex t. Then $[v, s, P, t, v]$ is a Hamiltonian cycle in G'.

$\Leftarrow$: In the other case, if C is the Hamiltonian cycle in G', then removing v from C will return a Hamiltonian path in G. ■

3. The low-degree spanning tree problem is as follows. Given a graph G and an integer k, does G contain a spanning tree such that all vertices in the tree have degree at most k (obviously, only tree edges count towards the degree)? For example, in the following graph, there is no spanning tree such that all vertices have a degree at most three.

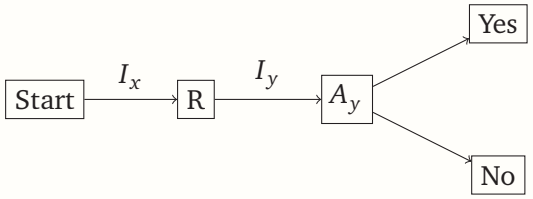


(a) Prove that the low-degree spanning tree problem is NP-hard with a reduction from Hamiltonian path.

Solution: We can prove this problem is NP-hard by a reduction from the Hamiltonian path problem. A Hamiltonian path visits every vertex exactly once, each vertex in the Hamiltonian path has atmost a degree of 2. This is because there is an incoming edge and an outgoing edge for each vertex in the path, except for the starting and ending vertices. So the hamiltonian path is nothing but a spanning tree of degree 2.

The equality= Hamiltonian Path $\leq$ Low-Degree spanning tree

Reduction figure for A_x:



Let G be the given Graph. For the above we have the following:

1) $I_x = G$

2) $I_y = G, k = 2$

3) R : G = (V,E)

4) A_x= Hamiltonian Path

5) A_y = Low-Degree spanning tree

The hamiltonian path is fed a graph G, and we feed the same graph and k as 2 to the Low-degree spanning tree. If the answer is Yes, then there is a hamiltonian path in the graph G. Thus we can say that the low-degree spanning tree problem is NP-hard. ■

(b) Now consider the high-degree spanning tree problem, which is as follows. Given a graph G and an integer k, does G contain a spanning tree whose highest degree vertex is at least k? In the previous example, there exists a spanning tree with a highest degree of 7. Give an efficient algorithm to solve the high-degree spanning tree problem, and an analysis of its time complexity.

Solution: The high-degree spanning tree problem is finding if the given graph G contains a spanning tree whose degree is ≥ k. We can solve it using the following algorithm.

```
HighSDP(G(V,E),k):
  visited= DFS(G)          ⟨⟨DFS or BFS to get visited vertices ⟩⟩
  if |visited|!=|V|
      return False
  for v in V
      degree[v] = degree of v
  if max(degree) < k
      return False
  return True
```

A graph can contain a spanning tree only if the graph is connected. We can check if the graph is connected by using DFS or BFS to get the visited vertices. If the number of vertices visited is equal to the number of vertices in the graph then the graph is connected. If it is not, we return "False". If "True", we find the degree of each vertex in the graph G and check if the highest degree vertex has a degree of at least k. If "False", the graph G doesn't contain a spanning tree whose highest degree vertex is atleast k. If "True", the graph G contains a spanning tree whose highest degree vertex is atleast k. Because we can always construct a spanning tree such that all the edges of the highest degree vertex are preserved and thus the resulting spanning tree's highest degree vertex will have a degree of atleast k.

The time complexity of running a DFS or BFS is O(V+E) and to find the degree of all the vertices we have to iterate through all the edges, thus time complexity will be O(V+E). So, the resulting time complexity is O(V+E). ■

4. Some SAT reductions:

   (a) Stingy SAT is the following problem: given a set of clauses (each a disjunction of literals) and an integer k, find a satisfying assignment in which at most k variables are true, if such an assignment exists. Prove that stingy SAT is NP-hard.

Solution: To prove that Stingy SAT is NP-hard, we need to reduce a known NP-hard problem to it. In this case, we will reduce the original Boolean satisfiability problem (SAT) to Stingy SAT.

Transformation: Given a SAT formula f with n variables, we will choose (f, n) as the instance of Stingy SAT.

We have to now show that f is a yes-instance of SAT iff (f, n) is a yes-instance of Stingy SAT.

- Suppose that f is a yes-instance of SAT. No more than n variables can be true, because there are a total of n variables. So any satisfying assignment of instance f for SAT will be a satisfying assignment of instance (f, n) for Stingy SAT. So (f, n) is a yes-instance of Stingy SAT.
- Suppose that (f, n) is a yes-instance of Stingy SAT. Any satisfying assignment of that instance will also be a satisfying assignment of instance f for SAT.So f is a yes-instance of SAT.

■

   (b) The Double SAT problem asks whether a given satisfiability problem has at least two different satisfying assignments. For example, the problem $\{\{v_1, v_2\}, \{\overline{v_1}, v_2\}, \{\overline{v_1}, \overline{v_2}\}\}$ is satisfiable, but has only one solution $(v_1 = F, v_2 = T)$. In contrast, $\{\{v_1, v_2\}, \{\overline{v_1}, \overline{v_2}\}\}$ has exactly two solutions. Show that Double-SAT is NP-hard.

Solution: To show that Double-SAT is NP-hard, we will reduce the Boolean satisfiability problem (SAT) to Double-SAT. This will demonstrate that if we had a polynomial-time algorithm for Double-SAT, we could use it to solve SAT, which is known to be NP-hard.

For an input of $\phi(x_1, x_2 ... x_n)$, let us introduce a new variable y such that the output formula becomes $\phi'(x_1, x_2 ... x_n, y) = \phi(x_1, x_2 ... x_n) \wedge (y \vee y')$

- If ϕ belongs to SAT, then it has at least one satisfying assignment and ϕ' has at two satisfying assignments by assigning y=T or y=F, such that $\phi' \in$ Double SAT. Similarly if $\phi \notin$ SAT, then ϕ' does not have a satisfying assignment either.
- If $\phi'(x_1, x_2 ... x_n, y) = \phi(x_1, x_2 ... x_n) \wedge (y \vee y')$ has a satisfiable assignment, then ϕ will also have a satisfiable assignment in SAT.

■