Graph Theory

BPM136

目录

Eulerian Tours

Introduction to Graph Models

Definition 1.1 : A **graph** G consists of two sets: V(G), called the vertex set, and E(G), called the edge set. An **edge**, denoted xy, is an unordered pair of vertices. We will often use G or G = (V, E) as short-hand.

Definition 1.2 : Let G be a graph.

- If xy is an edge, then x and y are the **endpoints** for that edge. We say x is **incident** to edge e if x is an endpoint of e.
- If two vertices are incident to the same edge, we say the vertices are **adjacent**. Similarly, if two edges share an endpoint, we say they are adjacent. If two vertices are adjacent, we say they are **neighbors** and the set of all neighbors of a vertex x is denoted N(x).
- If a vertex is not incident to any edge, we call it an **isolated vertex**.
- If both endpoints of an edge are the same vertex, then we say the edge is a **loop**.
- If there is more than one edge with the same endpoints, we call these **multi-edges**.
- If a graph has no multi-edges or loops, we call it **simple**.
- The **degree** of a vertex is the number of edges incident to that vertex, with a loop adding two to the degree. Denote the degree of vertex v as deg(v). If the degree is even, the vertex is valled **even**; also is **odd**.

Touring a Graph

Definition 1.3 : Let G be a graph.

- A **walk** is a sequence of vertices so that there is an edge between consecutive vertices. A walk can repeat verrtices and edges.
- A **trail** is a walk with no repeated edges. A trail can repeat vertices but not edges.
- A **path** is a trail with no repeated vertex(or edges). A path on n vertices is denoted P_n.
- A **closed walk** is a walk that starts and ends at the same vertex.
- A **circuit** is a closed trail; that is, a trail that starts and ends at the same vertex with no repeated edges though vertices may be repeated.
- A **cycle** is a closed path; that is, a path that starts and ends at the same vertex. Thus cycles cannot repeat edges or vertices. Note: we do not consider the starting and ending vertex as being repeated since each vertex is entered and exited exactly once. A cycle in n vertices is denoted C_n.

The **Length** of any of these tours is defined in terms of the number of edges. For example, P_n has length $n-1$ and C_n has length n.

Defnition 1.4 : Let G be a graph. Two vertices x and y are **conneccted** if there exists a path from x to y in G. The graph G is **connected** if every pair of distinct vertices is connected.

Defnition 1.5 : Let G be a graph. An **Eulerian circuit** (or **trail**) is a circuit (or trail) that contains every edge and every vertex of G. If G contaions an Eulerian circuit it is called **Eulerian** and if G contains an Eulerian tail but not an Eulerian circuit is is called **semi-Eulerian**.

Defnition 1.6 : A graph G is Eulerian iff

1. G is connected and
2. every vertex is even.

A graph G is semi-Eulerian iff

1. G is connected and
2. exactly two vertices is odd.

Theorem 1.7 (Handshaking Lemma) : Let G = (V, E) be a graph and $|E|$ denote the number of edges in G. Then the sum of the degrees of the vertices equals twice the number of edges; that is if $V = \{v_1, v_2, ..., v_n\}$, then

$$deg(v_1) + deg(v_2) + ... + deg(v_n) = 2|E|$$

Corollary 1.8 : There must be an even number of odd vertices in any graph G.

Eulerian Circuit Algorithms

Fleuery's Algorithm

Input: Connected graph G where zero or two vertices are odd.

Steps:

1. Choose a starting vertex, call it v. If G has no odds, then any vertex can be starting point. If G has exactly two odds, then v must be one of the odd.
2. Choose an edge incident to v that is unlabeled and label it with the number in which it was chosen, ensuring that the graph consisting of unlabeled edges remains connected.
3. Travel along the edge to its other endpoints.
4. Repeat steps 2 and 3 until all edges have been labeled.

Output: Labeled Eulerian circuit or trail.

Hierholzer's Algorithm

Input: Connected graph G where all vertices are even.

Steps:

1. Choose a starting vertex, call it v. Find a circuit C originating at v.

2. If any vertex x on C has edges not appearing in C, find a circuit $C^{'}$ originating at x that uses two of these edges.

3. Combine C and $C^{'}$ into a single circuit $C^{''}$

4. Repeat steps 2 and 3 until all edges of G are used.

Output: Labeled Eulerian circuit.

Eulerization

Definition 1.10 : Given a connected graph G = (V, E), an **Eulerization** of G is the graph $G^{'} = (V, E^{'})$ so that

(i) $G^{'}$ is obtained by duplicating edges of G, and

(ii) every vertex of $G^{'}$ is even.

A **semi-Eulerization** of G results in a graph $G^{'}$ so that

(i) $G^{'}$ is obtained by duplicating edges of G, and

(ii) exactly two vertices of $G^{'}$ are odd.

Eulerization Method

1. Identify the odd vertices of the graph.

2. Pair up the odd vertices, trying to pair as many adjacent vertices as possible while also avoiding pairing vertices far away from each other.

3. Duplicate the edges along an optimal path from one vertex to its pair.

Chinese Postman Problem

Definition 1.11 : **A weighted graph** G = (V, E, w) is a graph where each of the edges has a real number associated with it. This number is referred to as the **weight** and denoted w(xy) for an edge xy.

So, The weighted version of an Eulerization problem is called the Chinese Postman Problem.

The choice of which edges to duplicate when working with a weighted graph relies in part on shortest paths between two vertices. The difficulty is in choosing which vertices to pair. This will be revisited in Chapter 3 when an algorithm for finding a shortest path is introduced.

Hamiltonian cycles

Definition 2.1 : A cycle in a graph G that contains every vertex of G is called a **Hamiltonian cycle**. A path that contains every vertex is called a **Hamiltonian path**.

Conditions for Existence

Recall from Chapter 1 that if a graph has an Eulerian circuit, then it cannot have an Eulerian trail, and vice versa. The same is not true for the Hamiltonian version:

- If a graph has a Hamiltonian cycle, it automatically has a Hamiltonian path (just leave off the last edge of the cycle to obtain a path).
- If a graph has a Hamiltonian path, it may or may not have a Hamiltonian cycle.

a few necessary conditions can be placed on the graph to ensure a Hamiltonian cycle is possible. These include, but are not limited to, the following:

(1) G must be connected.
(2) No vertex of G can have degree less than 2.
(3) G cannot contain a **cut-vertex**, that is a vertex whose removal disconnects the graph.

Theorem 2.2 (Dirac's Theorem) : Let G be a graph with $n \geq 3$ vertices. If every vertex of G satisfies $deg(v) \geq \frac{n}{2}$, then G has a Hamiltonian cycle.

Hints : The proof of this theorem boils down to the fact that each vertex has so many edges incident to it that in trying to find a cycle we will never get stuck at a vertex. But this property is not a necessary condition.

Definition 2.3 : A simple graph G is complete if every pair of distinct vertices is adjacent. The complete graph on n vertices is denoted K_n.

Theorem 2.5 : Given a specified reference point, the complete graph K_n has $(n1)!$ distinct Hamiltonian cycles. Half of these cycles are reversals of the others.

Traveling Salesman Problem

How should a delivery service plan its route through a city to ensure each customer is reached? Historically, the extensive study of Hamiltonian circuits arose in part from a similar question: A traveling salesman has customers in numerous cities. He must visit each of them and return home, but wishes to do this with the least total cost. Determine the cheapest route possible for the salesman.

Brute Force Algorithm

Input: Weighted complete graph K_n.

Steps:

1. Choose a starting vertex, call it v.
2. Find all Hamiltonian cycles starting at v. Calculate the total weight of each cycle.
3. Compare all (n − 1)! cycles. Pick one with the least total weight. (Note: there should be at least two options).

Output: Minimum Hamiltonian cycle.

Nearest Neighbor Algorithm

Input: Weighted complete graph K_n.

Steps:

1. Choose a starting vertex, call it v. Highlight v.

2. Among all edges incident to v, pick the one with the smallest weight. If two possible choices have the same weight, you may randomly pick one.

3. Highlight the edge and move to its other endpoint u. Highlight u.

4. Repeat steps (2) and (3), where only edges to unhighlighted vertices are considered.

5. Close the cycle by adding the edge to v from the last vertex highlighted. Calculate the total weight.

Output: Hamiltonian cycle.

Repetitive Nearest Neighbor Algorithm

Input: Weighted complete graph K_n.

Steps:

1. Choose a starting vertex, call it v.

2. Apply the Nearest Neighbor Algorithm.

3. Repeat steps (1) and (2) so each vertex of Kn serves as the starting vertex.

4. Choose the cycle of least total weight. Rewrite it with the desired reference point.

Output: Hamiltonian cycle.

Cheapest Link Algorithm

Input: Weighted complete graph K_n.

Steps:

1. Among all edges in the graph pick the one with the smallest weight. If two possible choices have the same weight, you may randomly pick one. Highlight the edge.

2. Repeat step (1) with the added conditions:

(a) no vertex has three highlighted edges incident to it; and

(b) no edge is chosen so that a cycle closes before hitting all the vertices.

3. Calculate the total weight.

Output: Hamiltonian cycle.

Nearest Insertion Algorithm

Input: Weighted complete graph K_n.

Steps:

1. Among all edges in the graph, pick the one with the smallest weight. If two possible choices have the same weight, you may randomly pick one. Highlight the edge and its endpoints.

2. Pick a vertex that is closest to one of the two already chosen vertices. Highlight the new vertex and its edges to both of the previously chosen vertices.

3. Pick a vertex that is closest to one of the three already chosen vertices. Calculate the increase in weight obtained by adding two new edges and deleting a previously chosen edge. Choose the scenario with the smallest total. For example, if the cycle obtained from (2) was a − b − c − a and d is the new vertex that is closest to c, we calculate: w(dc) + w(db) − w(cb) and w(dc) + w(da) − w(ca) and choose the option that produces the smaller total.

4. Repeat step (3) until all vertices have been included in the cycle.

5. Calculate the total weight.

Output: Hamiltonian cycle

Definition 2.6 : The **relative error** for a solution is given by

$$\epsilon = \frac{SolutionOptimal}{Optimal}$$

Digraphs

Definition 2.7 : A **directed graph**, or **digraph**, is a graph G = (V, A) that consists of a vertex set V (G) and an arc set A(G). An arc is an ordered pair of vertices.

Definition 2.8 : Let G = (V, A) be a digraph.

- Given an arc xy, the head is the starting vertex x and the tail is theending vertex y.

  - a is the head of arc ab and the tail of arcs da and ba

- Given a vertex x, the in-degree of x is the number of arcs for which x is a tail, denoted $deg^{-}(x)$. The out-degree of x is the number of arcs for which x is the head, denoted $deg^{+}(x)$.
- The underlying graph for a digraph is the graph $G^{'} = (V, E)$ which is formed by removing the direction from each arc to form an edge.
- A directed path is a path in which the tail of an arc is the head of the next arc in the path.

  – d → a → b is a directed path, but c → d → a → b is not since cd is not an arc in the graph.

- A directed cycle is a cycle in which the tail of an arc is the head of the next arc in the cycle.

  – a → b → a is a directed cycle, but d → a → b → d is not.

Theorem 2.9 : Let G = (V, A) be a digraph and |A| denote the number of arcs in G. Then both the sum of the in-degrees of the vertices and the sum of the out-degrees equals the number of arcs; that is, if $V = \{v_1, v_2, ..., v_n\}$, then

$$deg(v_1) + ... + deg(v_n) = |A| = deg^{+}(v_1) + ... + deg^{+}(v_n)$$

Theorem 2.10 (A similar result for digragh of Dirac's theorem) : Let G be a digraph. If $deg(v) \geq \frac{n}{2}$ and $deg^{+}(v) \geq \frac{n}{2}$ for every vertex of G, then G has a Hamiltonian cycle.

Asymmetric Traveling Salesman Problem : The Asymmetric Traveling Salesman Problem is to find the optimal Hamiltonian directed cycle on a digraph in which the weight of arc ab need not equal the weight of arc ba.

Definition 2.10.1 : A **complete digraph** is similar to a complete graph with the added condition that between any two distinct vertices x and y both arcs xy and yx exist in the digraph.

Undirecting Algorithm

Input: Weighted complete digraph G = (V, A, w).

Steps:

1. For each vertex x make a clone $x^{'}$. Form the edge $xx^{'}$ with weight 0.
2. For each arc xy form the edge $x^{'}y$.
3. The weight of an edge is equal to the weight of its corresponding arc; that is

$$w(xx^{'}) = 0$$

$$w(x^{'}y) = w(xy)$$

$$w(xy^{'}) = w(yx)$$

.

Output: Weighted clone graph $G^{'} = (V^{'}, E^{'}, w)$, where $V^{'}$ consists of all vertices from G and their clones and $E^{'}$ the edges described above.

Paths

Shortest Paths

Dijkstra's Algorithm

Input: Weighted connected simple graph G = (V, E) and vertices designated as Start and End.

Steps:

1. For each vertex x of G, assign a label L(x) so that L(x) = (−, 0) if x = Start and L(x) = (−, ∞) otherwise. Highlight Start.

2. Let u = Start and define F to be the neighbors of u. Update the labels for each vertex v in F as follows:

$$if\ w(u) + w(uv) < w(v),\ then\ redefine\ L(v) = (u, w(u) + w(uv))$$

$$otherwise\ do\ not\ change\ L(v)$$

3. Highlight the vertex with lowest weight as well as the edge uv used to update the label. Redefine u = v.

4. Repeat steps (2) and (3) until the ending vertex has been reached. In all future iterations, F consists of the un-highlighted neighbors of all previously highlighted vertices and the labels are updated only for those vertices that are adjacent to the last vertex that was highlighted.

5. The shortest path from Start to End is found by tracing back from End using the first component of the labels. The total weight of the path is the weight for End given in the second component of its label.

Output: Highlighted path from Start to End and total weight w(End).

SPFA Algorithm

emmmm……. Maybe should be named "The Queue version of Bellman-Ford algorithm"

Input: Weighted connected simple graph G = (V, E) and vertices designated as Start and End.

Steps:

1. For each vertex x of G, assign a label L(x) so that L(x) = (−, 0) if x = Start and L(x) = (−, ∞) otherwise. Highlight Start. And push Start to a Queue Q.

2. Let u = the front of Q and define F to be the neighbors of u. update the labels for each vertex v in F as follows:

$$if\ w(u)+w(uv) < w(v),\ then\ redefine\ L(v) = (u, w(u)+w(uv))\ and,\ if\ v\ is\ not\ to\ be\ highlight\ then\ push\ v\ in\ to\ Q$$

$$otherwise\ do\ not\ change\ L(v)$$

3. Let u be unhighlight.

4. Repeat steps 2, 3 until the size of Q is 0.

Output: Total weight w(End).

Trees and Networks

Trees

Definition 4.1 : A graph G is

- **acyclic** if there are no cycles or circuits in the graph.
- a **network** if it is connected.
- a **tree** if it is an acyclic network; that is, a graph that is both acyclic and connected.
- a **forest** if it is an acyclic graph.

In addition, a vertex of degree 1 is called a **leaf**.

Properties of Trees

(1) For every $n \geq 1$, any tree with n vertices has n − 1 edges.

(2) For any tree with $n \geq 1$ vertices, the sum of the degrees is 2n − 2.

(3) Every tree with at least two vertices contains at least two leaves.

(4) Any network on n vertices with n − 1 edges must be a tree.

(5) For any two vertices in a tree, there is a unique path between them.

(6) The removal of any edge of a tree will disconnect the graph.

Spanning Trees

Definition 4.2 A **subgraph** H of a graph G is a graph where H contains some of the edges and vertices of G; that is, $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

We say H is a **spanning subgraph** if it contains all the vertices but not necessarily all the edges of G; that is, $V(H) = V(G)$ and $E(H) \subseteq E(G)$.

A **spanning tree** is a spanning subgraph that is also a tree.

Kruskal's Algorithm

Input: Weighted connected graph G = (V, E).

Steps:

1. Choose the edge of least weight. Highlight it and add it to $T = (V, E^{'})$.

2. Repeat step (1) so long as no circuit is created. That is, keep picking the edges of least weight but skip over any that would create a cycle in T.

Output: Minimum spanning tree T of G.

Prim's Algorithm

Input: Weighted connected graph G = (V, E).

Steps:

1. Let v be the root. If no root is specified, choose a vertex at random. Highlight it and add it to $T = (V^{'}, E^{'})$.

2. Among all edges incident to v, choose the one of minimum weight. Highlight it. Add the edge and its other endpoint to T.

3. Let S be the set of all edges with exactly endpoint from V (T). Choose the edge of minimum weight from S. Add it and its other endpoint to T.

4. Repeat step (3) until T contains all vertices of G, that is V (T) = V (G).

Output: Minimum spanning tree T of G.

Borůvka's Algorithm

Input: Weighted connected graph G = (V, E) where all the weights are distinct.

Steps:

1. Let T be the forest where each component consists of a single vertex.

2. For each vertex v of G, add the edge of least weight incident to v to T.

3. If T is connected, then it is a minimum spanning tree for G. Otherwise, for each component C of T, find the edge of least weight from a vertex in C to a vertex not in C. Add the edge to T.

4. Repeat Step (3) until T has only one component, making T a tree.

Output: A minimum spanning tree for G.

Traveling Salesman Problem Revisted

Definition 4.3.1 : Traveling Salesman Problem (mTSP) only considers scenarios where the weights satisfy the **triangle inequality**; that is, for a weighted graph G = (V, E, w), given any three vertices x, y, z we have $w(xy) + w(yz) \geq w(xz)$.

mTSP Algorithm

Input: Weighted complete graph K_n, where the weight function w satisfies the triangle inequality.

Steps:

1. Find a minimum spanning tree T for K_n.
2. Duplicate all the edges of T to obtain T^*.
3. Find an Eulerian circuit for T^*.
4. Convert the Eulerian circuit into a Hamiltonian cycle by skipping any previously visited vertex (except for the starting and ending vertex).
5. Calculate the total weight.

Output: Hamiltonian cycle for K_n.

Matching

Definition 5.1 : Given a graph G = (V, E), a **matching** M is a subset of the edges of G so that no two edges share an endpoint. When two edges do not share an endpoint, we call them **independent** edges. The size of a matching, denoted |M|, is the number of edges in the matching.

Bipartile Graphs

Definition 5.2 : A graph G = (V, E) is **bipartite** if the vertices can be partitioned (or split) into two sets, X and Y , so that X and Y have no vertices in common, every vertex appears in either X or Y , and every edge has exactly one endpoint in X and the other endpoint in Y . We denote this by G = (X Y, E).

Theorem 5.3 : A graph G is bipartite iff there are no odd cycles in G.

Definition 5.4 : A simple bipartite graph G = (X Y, E) is a **complete bipartite graph** if every vertex in X is adjacent to every vertex in Y . If the size of X is m and the size of Y is n, then we write $K_{m,n}$.

Matching Terminology and Strategies

Definition 5.5 : A vertex is **saturated** by a matching M if it is incident to an edge of the matching; otherwise, x is called **unsaturated**.

Definition 5.6 : Given a matching M on a graph G, we say M is

- **maximal** if M cannot be enlarged by adding an edge.
- **maximum** if M is of the largest size amongst all possible matchings.
- **perfect** if M saturates every vertex of G.
- an **X-matching** if it saturates every vertex from the collection of vertices X (a similar definition holds for a Y matching).

Theorem 5.7 (Hall's Marriage Theorem) : Given a bipartite graph $G = (X \cup Y, E)$, there exists an X-matching if and only if any collection S of vertices from X satisfies $|S| \leq |N(S)|$.

Augmenting Paths and Vertex Covers

Definition 5.8 : Given a matching M of a graph G, a path is called

- **M-alternating** if the edges in the path alternate between edges that are part of M and edges that are not part of M.
- **M-augmenting** if it is an M-alternating path and both endpoints of the path are unsaturated by M, implying both the starting and ending edges of the path are not part of M.

Theorem 5.9 (Berge's Theorem) A matching M of a graph G is maximum iff G does not contain any M-augmenting paths.

Augmenting Path Algorithm
Input: Bipartite graph $G = (X \cup Y, E)$.
Steps:

1. Find an arbitrary matching M.
2. Let U denote the set of unsaturated vertices in X.
3. If U is empty, then M is a maximum matching; otherwise, select a vertex x from U.
4. Consider y in N(x).
5. If y is also unsaturated by M, then add the edge xy to M to obtain a larger matching $M^{'}$. Return to Step 2 and recompute U. Otherwise, go to Step 6.
6. If y is saturated by M, then find a maximal M-alternating path from x using xy as the first edge.

(a) If this path is M-augmenting, then switch edges along that path to obtain a larger matching $M^{'}$; that is, remove from M the matched edges along the path and add the unmatched edges to create $M^{'}$. Return to Step (2) and recompute U.

(b) If the path is not M-augmenting, return to Step (4), choosing a new vertex from N(x).

7. Stop repeating Steps (2) – (4) when all vertices from U have been considered.

Output: Maximum matching for G

Definition 5.10 : A **vertex cover** Q for a graph G is a subset of vertices so that every edge of G has at least one endpoint in Q.

Theorem 5.11 (*$\widetilde{K}onig-Egervary\ Theorem$*) For a bipartite graph G, the size of a maximum matching of G equals the size of a minimum vertex cover for G.

Vertex Cover Method

1. Let $G = (X \cup Y, E)$ be a bipartite graph.
2. Apply the Augmenting Path Algorithm and mark the vertices considered throughout its final implementation.
3. Define a vertex cover Q as the unmarked vertices from X and the marked vertices from Y.
4. Q is a minimum vertex cover for G.

Definition 5.12 : Given a collection of finite nonempty sets $S_1, S_2, ..., S_n$ (where $n \geq 1$), a system of **distinct representatives** is a collection $r_1, r_2, ..., r_n$ so that r_i is a member of set S_i and $ri \neq rj$ for all $i \neq j$ (for all $i, j = 1, 2, ..., n$).

Connected component

Strongly Connected component

Definition 6.1 : a graph $G = (V, E)$ is said to be **strongly connected** or **diconnected** if every vertex is reachable from every other vertex.

Definition 6.2 : The **strongly connected components** or **diconnected components** of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected.

Kosaraju's algorithm

Input: a graph $G = (V, E)$.

Steps:

1. For each vertex u of the graph, mark u as unvisited. Let an ordered list L be empty.
2. For each vertex u of the graph do Visit(u), where Visit(u) is the recursive subroutine:

- If u is unvisited then:
  - Mark u as visited.

  - For each out-neighbour v of u, do Visit(v).
  - Prepend u to L.

- Otherwise do nothing.

3. For each element u of L in order, do Assign(u,u) where Assign(u,root) is the recursive subroutine:

- If u has not been assigned to a component then:
  - Assign u as belonging to the component whose root is root.
  - For each in-neighbour v of u, do Assign(v,root).

Otherwise do nothing.
Output: Some strongly connected components of G.

Biconnected component

Definition 6.3 : a **biconnected component** or **2-connected component** or **2-point-connected component** is a maximal biconnected subgraph. Any connected graph decomposes into a tree of biconnected components called the **block-cut tree** of the graph. The blocks are attached to each other at shared vertices called **cut vertices** or **articulation points**. Specifically, a **cut vertex** is any vertex whose removal increases the number of connected components.

Definition 6.4 : a **bridge**, **isthmus**, **cut-edge**, or **cut arc** is an edge of a graph whose deletion increases its number of connected components. Equivalently, an edge is a bridge if and only if it is not contained in any cycle. A graph is said to be **bridgeless** or **isthmus-free** if it contains no bridges.

Tarjan's articulation-point-finding algorithm

```
GetArticulationPoints(i, d)
    visited[i] := true
    depth[i] := d
    low[i] := d
    childCount := 0
    isArticulation := false

    for each ni in adj[i] do
        if not visited[ni] then
            parent[ni] := i
            GetArticulationPoints(ni, d + 1)
            childCount := childCount + 1
            if low[ni] >= depth[i] then
                isArticulation := true
            low[i] := Min(low[i], low[ni])
        else if ni != parent[i] then
            low[i] := Min(low[i], depth[ni])
    if (parent[i] != null and isArticulation) or (parent[i] = null and childCount > 1) then
        Output i as articulation point
```

Tarjan's bridge-finding algorithm
Input: a Graph G = (V, E)
Steps:

- 1. Find a spanning forest of G.
- 2. Create a rooted forest F from the spanning tree.
- 3. Traverse the forest F in preorder and number the nodes. Parent nodes in the forest now have lower numbers than child nodes.
- 4. For each node v in preorder (denoting each node using its preorder number), do:
    - Compute the number of forest descendants ND(v) for this node, by adding one to the sum of its children's descendants.
    - Compute L(v), the lowest preorder label reachable from v by a path for which all but the last edge stays within the subtree rooted at v. This is the minimum of the set consisting of the preorder label of v, of the values of L(w) at child nodes of v and of the preorder labels of nodes reachable from v by edges that do not belong to F.
    - Similarly, compute H(v), the highest preorder label reachable by a path for which all but the last edge stays within the subtree rooted at v. This is the maximum of the set consisting of the preorder label of v, of the values of H(w) at child nodes of v and of the preorder labels of nodes reachable from v by edges that do not belong to F.
    - For each node w with parent node v, if L(w)=w and H(w) < w + ND(w) then the edge from v to w is a bridge.