Chap 4. Spanning Trees

Yih-Lang Li (李毅郎)

Computer Science Department

National Chiao Tung University, Taiwan

The sources of most figure images are from the course slides (Graph Theory) of Prof. Gross

Outline

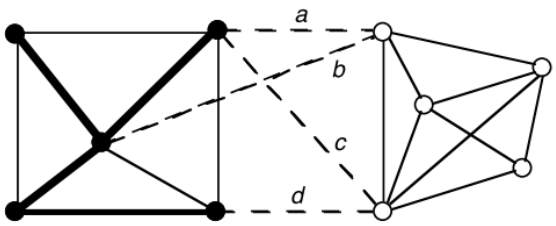
- Tree Growing
- Depth-First and Breadth-First Search
- Minimum Spanning Trees and Shortest Paths
- Application of Depth-First Search
- Cycles, Edge-Cuts, and Spanning Trees
- Graphs and Vector Spaces

4.1 Tree Growing

- **TERMINOLOGY**: For a given tree T in a graph G, the edges and vertices of T are called ***tree edges*** and ***tree vertices***, and the edges and vertices of G that are not in T are called ***non-tree edges*** and ***non-tree vertices***.

Frontier Edges

- **DEFINITION**: A ***frontier edge*** for a given tree T in a graph is a non-tree edge with one endpoint in T, called its ***tree endpoint***, and one endpoint not in T, its ***non-tree endpoint***.
- **Example 4.1.1.**

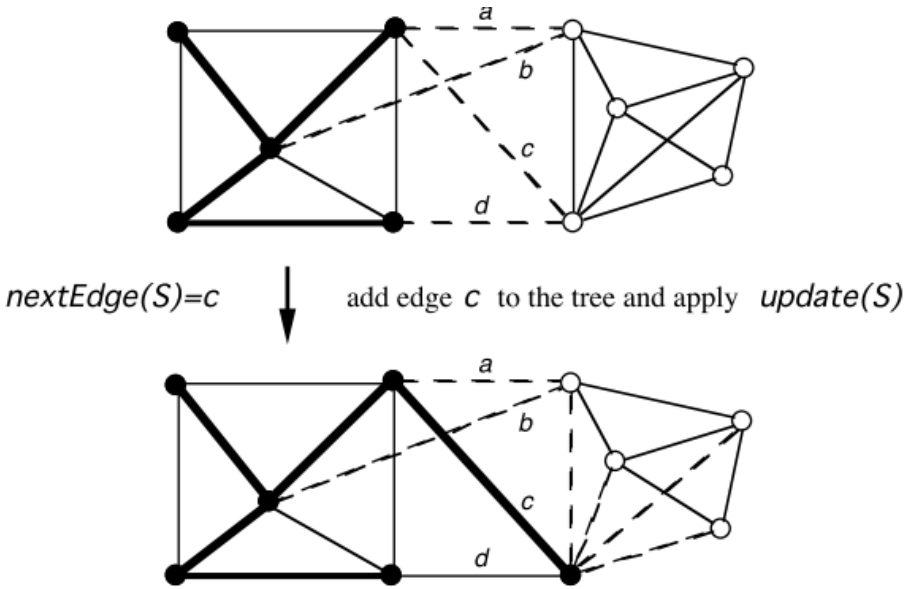


A tree with frontier edges a, b, c, and d.

- **Proposition 4.1.1.** *Let T be a tree in a graph G, and let e be a frontier edge for T. Then the subgraph of G formed by adding edge e to tree T is a tree.*
  - ✓ Adding a frontier edge does not produce a cycle and subgraph is still connected.
- **Remark**: Formally, adding a frontier edge to a tree involves adding a new vertex to the tree, as well as the primary operation of *adding an edge*, defined in §2.4.

Choosing a Frontier Edge

- **DEFINITION**: Let T be a tree subgraph of a graph G, and let S be the set of frontier edges for T. The function ***nextEdge*(*G*,*S*)** chooses and returns as its value the frontier edge in S that is to be added to tree T.
- **Remark**: Ordinarily, the function *nextEdge* is deterministic; however, it may also be randomized. In either case, the full specification of *nextEdge* must ensure that it always returns a frontier edge.
- **DEFINITION**: After a frontier edge is added to the current tree, the procedure ***updateFrontier*(*G*,*S*)** removes from S those edges that are no longer frontier edges and adds to S those that have become frontier edges.
- **Example 4.1.1. continue.**



Result after adding edge c to the tree; note deletion of edge d from the frontier.

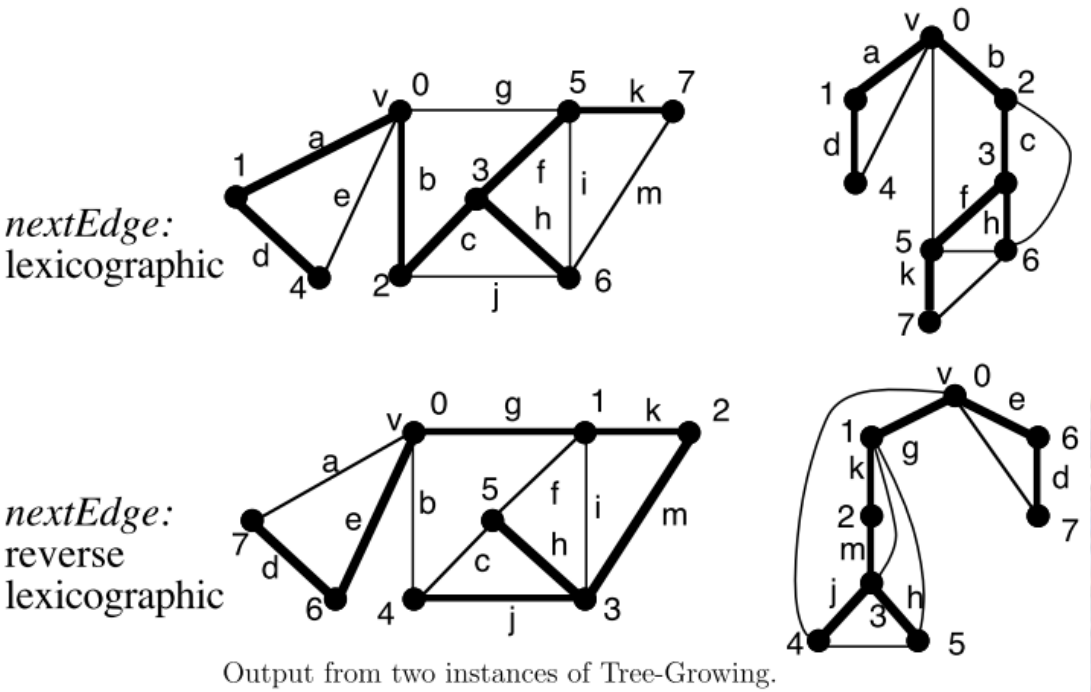
Choosing a Frontier Edge

- **Remark**: Each different version of *nextEdge*, based on how its selection rule is defined, creates a different instance of Tree-Growing.† In §4.2 and §4.3, we will see that the four classical algorithms, *depth-first search*, *breadth-first search*, *Prim's spanning-tree*, and *Dijkstra's shortest path*, are four different instances of Tree-Growing.

ALGORITHM: TREE-GROWING(G, v)
Input: a connected graph G, a starting vertex $v \in V_G$, and a selection-function $nextEdge$.
Output: an ordered spanning tree T of G with root v.
Initialize tree T as vertex v.
Initialize S as the set of proper edges incident on v.
While $S \neq \emptyset$
    Let $e = nextEdge(G, S)$.
    Let w be the non-tree endpoint of edge e.
    Add edge e and vertex w to tree T.
    $updateFrontier(G, S)$.
Return tree T.

Discovery Order of the Vertices

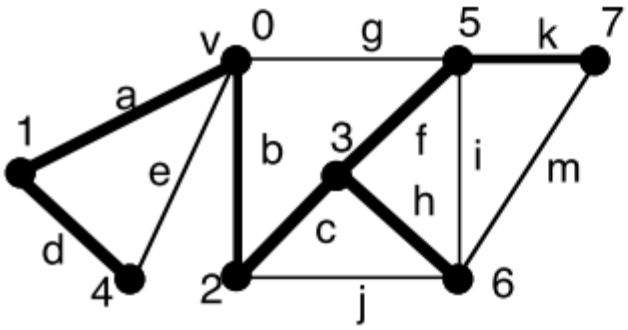
- **DEFINITION**: Let *T* be the spanning tree produced by any instance of Tree-Growing in a graph *G*. The ***discovery order*** is a listing of the vertices of *G* in the order in which they are added (discovered) as tree *T* is grown. The position of a vertex in this list, starting with 0 for the start vertex, is called the ***discovery number*** of that vertex.
- **Proposition 4.1.2.** *Let T be the output tree produced by any instance of Tree-Growing in a graph G. Then T is an ordered tree with respect to the discovery order of its vertices.*
- **Example 4.1.2.**
- **Example 4.1.3.**
  - ✓ if the tree produced by tree growing does not contain all edges of original graph *G*, *G* contains at least one cycle.
  - ✓ When every non-tree edge is added to the tree, a cycle is formed.

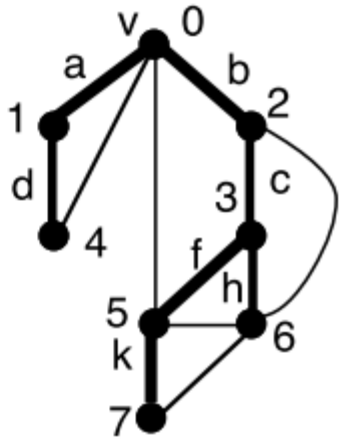


Output from two instances of Tree-Growing.

Two Kinds of Non-Tree Edges

- **DEFINITION**: Two vertices in a rooted tree are ***related*** if one is a descendant of the other.
- **DEFINITION**: For a given output tree grown by Tree-Growing, a ***skip-edge*** is a non-tree edge whose endpoints are related; a ***cross-edge*** is a non-tree edge whose endpoints are not related.
- **Example 4.1.3.** Edges *e*, *g*, and *j* are skip-edges, while edges *i* and *m* are cross-edges.



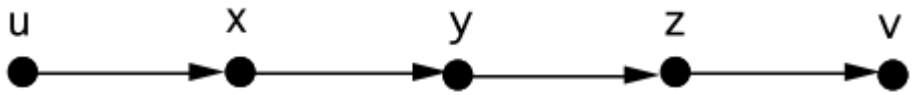


Tree-Growing Applied to a Non-Connected Graph

- **Proposition 4.1.3.** *Let T be the output tree produced when an instance of Tree-Growing (Algorithm 4.1.1) is applied to a graph G (not necessarily connected), starting at a vertex $v \in V_G$. Then T is a spanning tree of the component $C_G(v)$.*
  - ✓ Assume that it holds as $|T|=n$. As $|T|=n+1$, we can get T^* by deleting from T a leaf w and its edge e of T. Since $|T^*|=n$, T^* is a spanning tree of $C_{G-w}(v)$.
  - ✓ w is connected from v in T and G. Thus w and e must be in $C_G(v)$. Thus T is also a spanning tree of $C_G(v)$.
- **Corollary 4.1.4.** *A graph G is connected if and only if the output tree produced when an instance of Tree-Growing is applied to G is a spanning tree of G.*

Tree-Growing in a Digraph

- **DEFINITION**: A ***frontier arc*** for a rooted tree T in a digraph is an arc whose tail is in T and whose head is not in T.
- The ways to determine the connectivities of a directed graph and an undirected graph are different.
- The number of vertices in the spanning tree derived from the spanning process on a directed graph depends on the starting vertex.
- **Example 4.1.4.**



Output tree depends on the starting vertex.

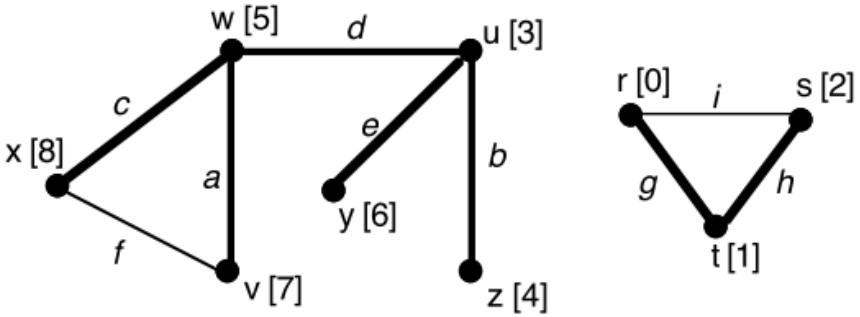
- **DEFINITION**: For tree-growing in digraphs, the non-tree edges (arcs) fall into three categories:
  - ✓ A **back-arc** is directed from a vertex to one of its ancestors.
  - ✓ A **forward-arc** is directed from a vertex to one of its descendants.
  - ✓ A **cross-arc** is directed from a vertex to another vertex that is unrelated.
- **TERMINOLOGY**: There are two kinds of cross-arcs: a ***left-to-right cross-arc*** is directed from smaller discovery number to larger one; a ***right-to-left*** is the opposite.

Forest Growing

- **DEFINITION**: A ***full spanning forest*** of a graph G is a spanning forest consisting of a collection of trees, such that each tree is a spanning tree of a different component of G.

ALGORITHM: FOREST-GROWING
Input: a graph G
Output: a full spanning forest F for G and the number $c(G)$.
Initialize forest F as the empty graph.
Initialize component counter $t := 1$
While forest F does not yet span graph G
    Let $v = nextVertex(V_G - V_F)$.
    By Tree-Growing, obtain spanning tree T_t of $C_G(v)$.
    Add tree T_t to forest F.
    $t := t + 1$
Return forest F and component count $c(G) = t$.

- **Example 4.1.5.**



Growing a 2-component spanning forest.

4.2 Depth-First and Breadth-First Search

- **TERMINOLOGY**: The output trees produced by the depth-first and breadth-first searches of a graph are called the ***depth-first tree*** (or ***dfs-tree***) and the ***breadth-first tree*** (or ***bfs-tree***).
- **TERMINOLOGY**: The use of the word "search" here is traditional. Although one does occasionally use these algorithms to search for something, they are tree-growing algorithms with much wider use.

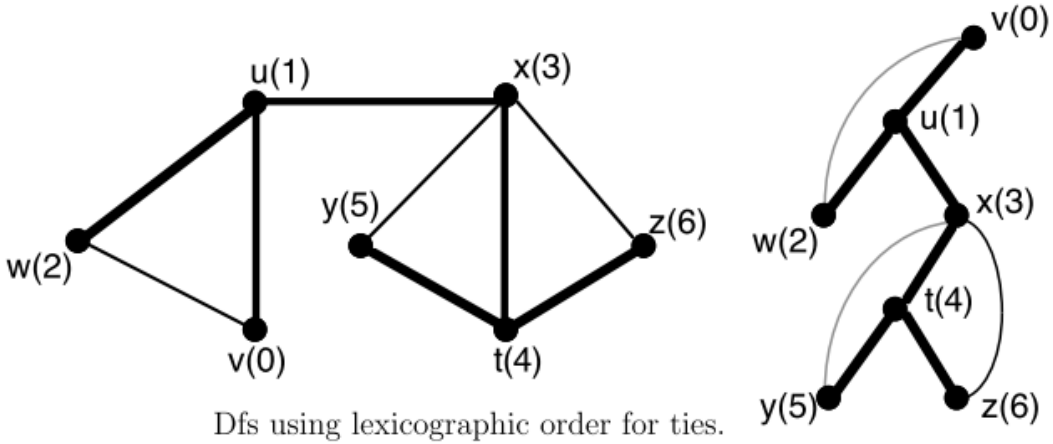
Depth-First Search

- **DEFINITION**: Let S be the current set of frontier edges. The function ***dfs-nextEdge*** is defined as follows: *dfs-nextEdge*(G,S) selects and returns as its value the frontier edge whose tree-endpoint has the largest discovery number. If there is more than one such edge, then *dfs-nextEdge*(G,S) selects the one determined by the default priority.

ALGORITHM: DEPTH-FIRST SEARCH
Input: a connected graph G, a starting vertex $v \in V_G$.
Output: an ordered spanning tree T of G with root v.
Initialize tree T as vertex v.
Initialize S as the set of proper edges incident on v.
While $S \neq \emptyset$
    Let $e = dfs\text{-}nextEdge(G, S)$.
    Let w be the non-tree endpoint of edge e.
    Add edge e and vertex w to tree T.
    $updateFrontier(G, S)$.
Return tree T.

Depth-First Search

- **Example 4.2.1.**



Dfs using lexicographic order for ties.

- **TERMINOLOGY**: The discovery number of each vertex w for a depth-first search is called the ***dfnumber*** of w and is denoted $dfnumber(w)$.
- **Proposition 4.2.1.** *Depth-first search trees have no cross-edge.*
  - ✓ For any non-tree edge e whose endpoints are x and y, we assume $dfnumber(x) < dfnumber(y)$.
  - ✓ When x is visited, e is an frontier edge but e is not selected as a tree edge.
  - ✓ Since e is finally a non-tree edge, we can see that y must has been added into the tree by another edge instead of e during the downward traverse process. Otherwise, e will be added into the tree when the traverse process backtracks to vertex x.
  - ✓ Thus vertex y must be a descendant of vertex x.

Depth-First Search

- **Remark:** A preorder traversal of the depth-first tree reproduces the discovery order generated by the depth-first search.

Recursive Depth-First Search

- **COMPUTATIONAL NOTE:** We mentioned in §4.1 that for tree-growing, the natural way to store the frontier edges is to use a *priority queue*. For the particular case of depth-first search, the priority queue emulate the simpler *stack* data structure (Last-In-First-Out), because ***the newest frontier edges are given the highest priority*** (***dfnumber is priority***) by being pushed onto the stack (in increasing default priority order if there is more than newest frontier edge). The recursive aspect of depth-first search also suggests the feasibility of implementation as a stack.

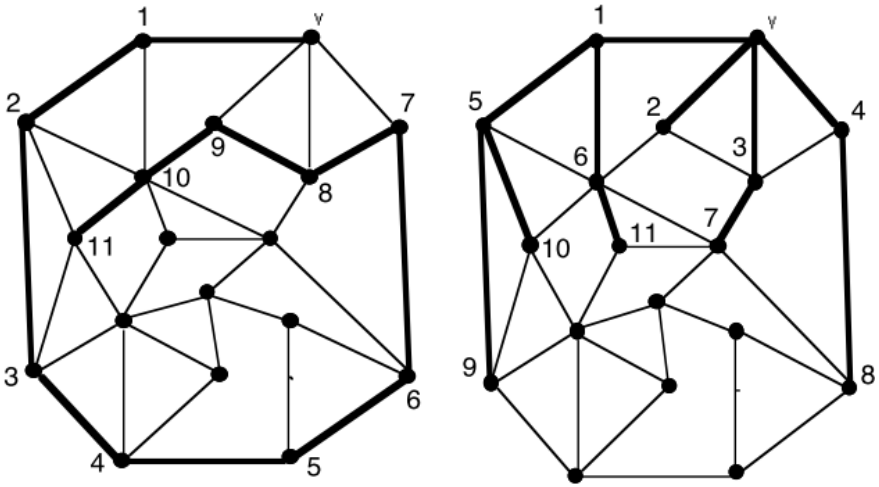
Depth-First Search in a Digraph

- **DEFINITION**: The function ***dfs-nextArc*** selects and returns as its value the frontier arc whose tree-endpoint has the largest dfnumber.
- **Proposition 4.2.2.** *When a depth-first search is executed on a digraph, the only kind of non-tree arc that cannot occur is a left-to-right cross arc.*

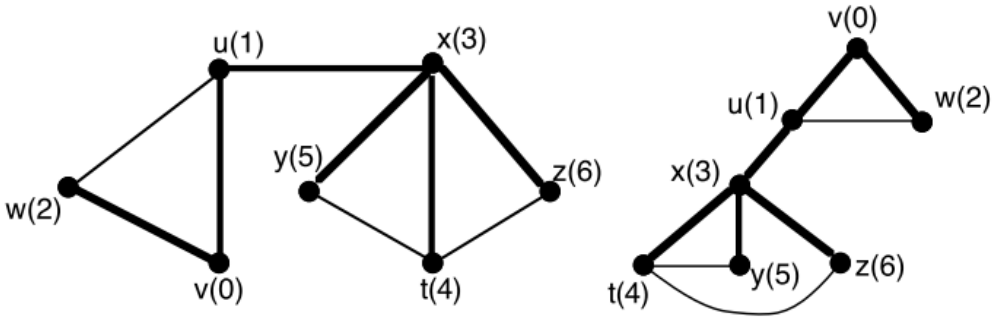
Breadth-First Search

- **Example 4.2.2:**
- **DEFINITION**: Let S be the current set of frontier edges. The function ***bfs-nextEdge*** is defined as follows: *bfs-nextEdge*(G,S) selects and returns as its value the frontier edge whose tree-endpoint has the smallest discovery number. If there is more than one such edge, then *bfs-nextEdge*(G,S) selects the one determined by the default priority.
- **Example 4.2.3:**

ALGORITHM: BREADTH-FIRST SEARCH
Input: a connected graph G, a starting vertex $v \in V_G$.
Output: an ordered spanning tree T of G with root v.
Initialize tree T as vertex v.
Initialize S as the set of proper edges incident on v.
While $S \neq \emptyset$
    Let $e = bfs\text{-}nextEdge(G, S)$.
    Let w be the non-tree (undiscovered) endpt of e.
    Add edge e and vertex w to tree T.
    $updateFrontier(G, S)$.
Return tree T.



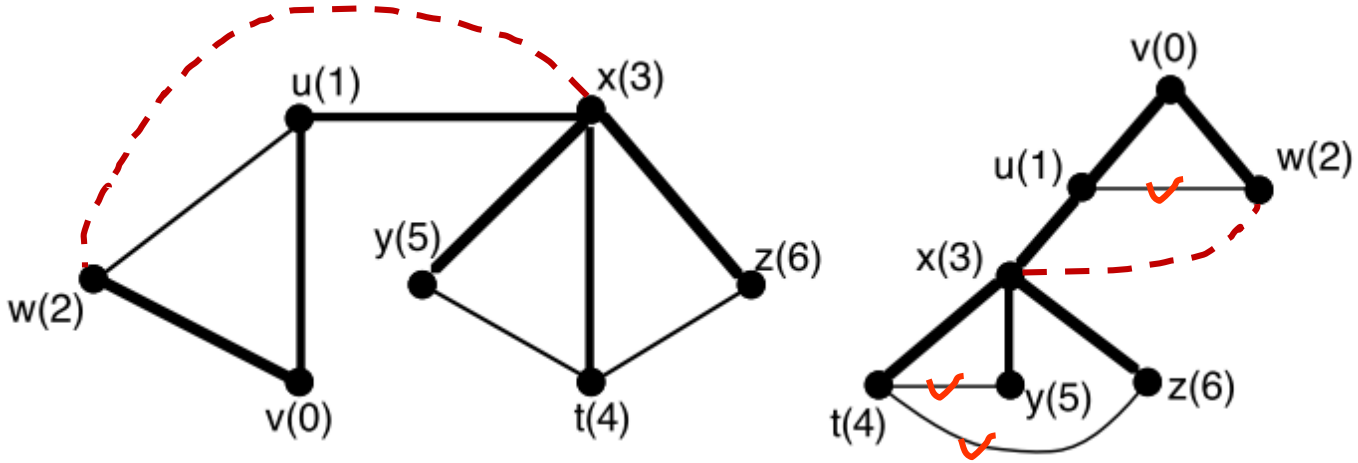
Dfs and bfs after 11 iterations.



Bfs using lexicographic order for ties.

Breadth-First Search

- **Proposition 4.2.3.** *When breadth-first search is applied to an undirected graph, the endpoints of each non-tree edge are either at the same level or at consecutive levels.*
- **Proposition 4.2.4.** *The breadth-first tree produced by any application of Algorithm 4.2.2 is a shortest-path tree for the input graph.*
- **COMPUTATIONAL NOTE:** Whereas the stack (Last-In-First-Out) is the appropriate data structure to store the frontier edges in a depth-first search, the queue (First-In-First-Out) is most appropriate for the breadth-first search, since the frontier edges that are oldest have the highest priority.

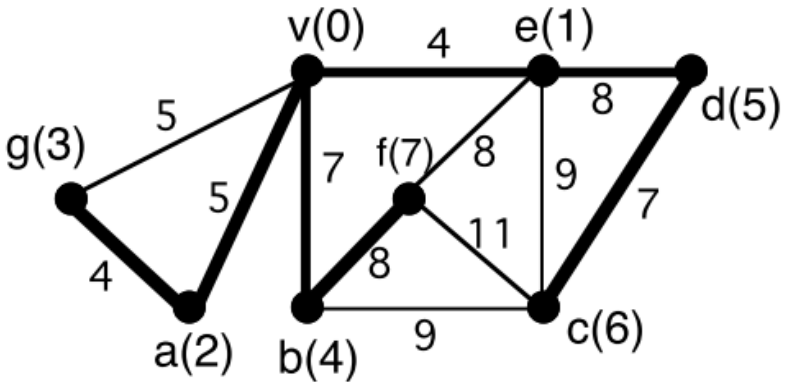


Bfs using lexicographic order for ties.

4.3 Minimum Spanning Tree and Shortest Paths

- *Minimum Spanning Tree Problem*: Let *G* be a connected weighted graph. Find a spanning tree of *G* whose total edge weight is minimum.
- **DEFINITION:** Let S be the current set of frontier edges. The function ***Prim-next Edge*** is defined as follows: *Prim-next Edge (G,S)* selects and returns as its value the frontier edge with smallest edge-weight. If there is more than one such edge, then *Prim-next edge (G,S)* selects the one determined by the default priority.

- **Example 4.3.1:**

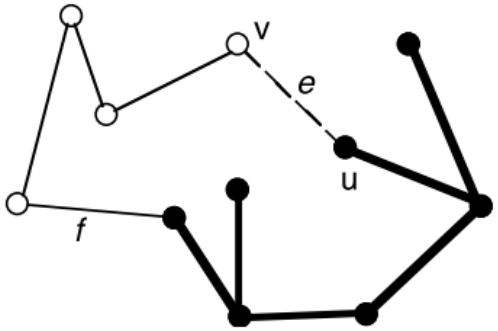


MST produced by Prim's algorithm.

ALGORITHM: PRIM MINIMUM SPANNING-TREE
Input: a weighted conn graph G and starting vertex v.
Output: a minimum spanning tree T.
Initialize tree T as vertex v.
Initialize S as the set of proper edges incident on v.
While $S \neq \emptyset$
    Let $e = Prim\text{-}nextEdge\,(G, S)$.
    Let w be the non-tree endpoint of edge e.
    Add edge e and vertex w to tree T.
    $updateFrontier(G, S)$.
Return tree T.

Finding the Minimum Spanning Tree: Prim's Algorithm

- **Proposition 4.3.1.** Let T_k be the Prim tree after k iterations of Prim's algorithm on a connected graph G, for $0 \le k \le |V_G| - 1$. Then T_k is a subtree of a MST of G.
  - ✓ Prove by induction. This is true as k=0 (T_0 is starting vertex).
  - ✓ Assume for some k, $0 \le k \le |V_G|$-2, T_k is a subtree of a minimum spanning tree T of G.
  - ✓ Algorithm produces T_{k+1} by adding a frontier edge e (from u to v) and new vertex v to T_k.
  - ✓ If T contains e, then T_{k+1} is a subtree of T.
  - ✓ If T does not contain e, then $T+e$ will form a cycle (in Fig). As we walk from v towards u along T, edge f is the first edge to enter T_k by connecting to x in T_k.
  - ✓ As T_k grows to T_{k+1}, Prim selects e instead of f, which implies $w(e) \le w(f)$.
  - ✓ $\widehat{T} = T - f + e$, and $w(\widehat{T}) \le w(T)$. Thus T is not a MST $\rightarrow$ contradiction.



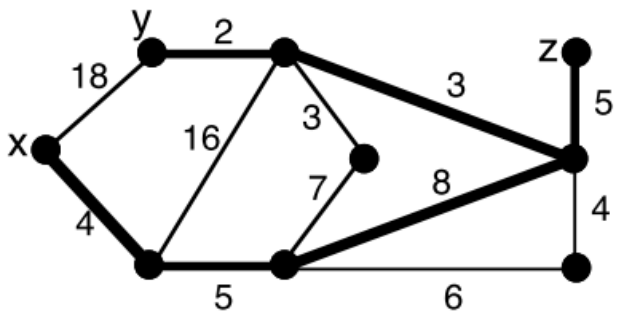
The tree T_k is in bold.

Finding the Minimum Spanning Tree: Prim's Algorithm

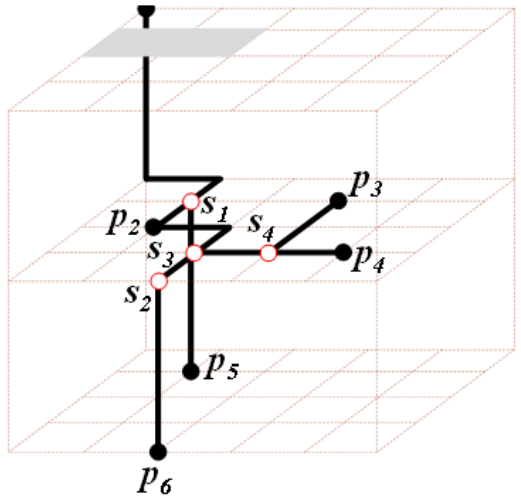
- **Corollary 4.3.2** When Prim's algorithm is applied to a connected graph, it produces a minimum spanning tree.
  - ✓ Proposition 4.3.1. implies that the Prim tree resulting from the final iteration is a MST.

The Steiner-Tree Problem

- **DEFINITION:** Let U be a subset of vertices in a connected edge-weighted graph G. The **Steiner-tree problem** is to find a minimum-weight tree subgraph of G that contains all the vertices of U.
- **Example 4.3.2:**



Steiner tree for $U = \{x, y, z\}$.



- **Remark:** The Steiner-tree problem often arises in network-design and wiring-layout problems.

Finding the Shortest Path: Dijkstra's Algorithm

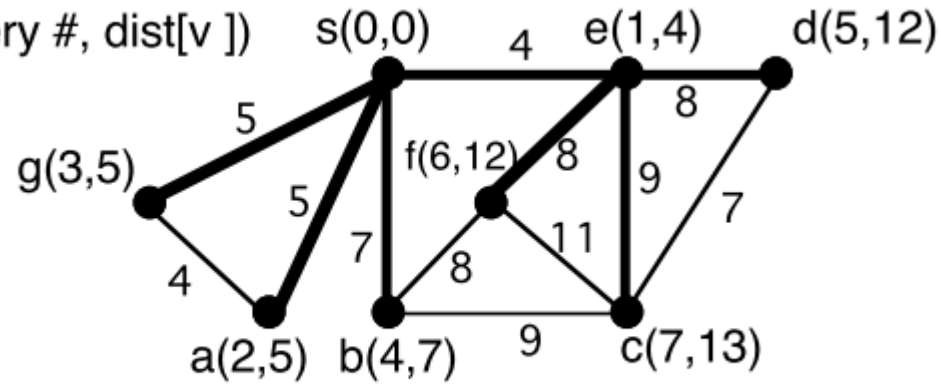
- *Shortest-Path Problem*: Let *s* and *t* be two vertices of a connected weighted graph. Find a path from *s* to *t* whose total edge-weight is minimum, i.e., a shortest *s*-*t* path.
- **Remark:** If the edge-weights are all equal, then the problem reduces to one that can be solved by breadth-first search (Algorithm 4.2.2).
- The algorithm finds a shortest path from vertex *s* to each of the vertices of the graph.
- **DEFINITION:** The function **Dijkstra-nextEdge** is defined as follows: Let *S* be the current set of frontier edges. *Dijkstra-nextEdge*(*G*,*S*) selects and returns as its value the frontier edge **whose non-tree endpoint is closest to the start vertex *s***. If there is more than one such edge, then *Dijkstra-nextEdge*(*G*,*S*) selects the one determined by the default priority.

ALGORITHM: DIJKSTRA SHORTEST PATH
Input: a weighted conn graph G and starting vertex s.
Output: a shortest-path tree T with root s.

Initialize tree T as vertex s.
Initialize S as the set of proper edges incident on s.
While $S \neq \emptyset$
    Let $e := Dijkstra\text{-}nextEdge\,(G, S)$.
    Let w be the non-tree endpoint of edge e.
    Add edge e and vertex w to tree T.
    $updateFrontier\,(G, S)$.
Return tree T.

Finding the Shortest Path: Dijkstra's Algorithm

- **NOTATION:** For each tree vertex *x*, let *dist*[*x*] denote the distance from vertex *s* to *x*.
- **Example 4.3.3:**



A Dijkstra shortest-path tree.

Calculating Distances as the Dijkstra Tree Grows

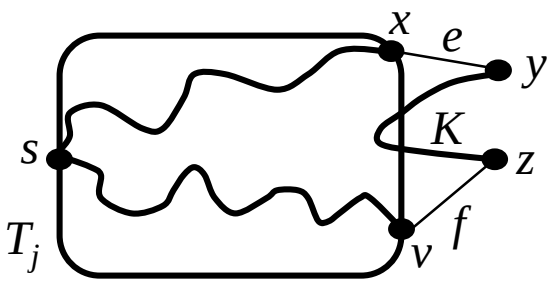
- **NOTATION:** For each frontier edge *e* in the weighted graph, *w*(*e*) denotes its edge-weight.
- **DEFINITION:** Let *e* be a frontier edge of the Dijkstra tree grown so far, and let *x* be the tree endpoint of *e*. The ***P-value*** of edge *e*, denoted *P*(*e*), is given by $P(e) = dis[x] + w(e)$

  Thus, *Dijkstra-nextEdge*(*G*,*S*) selects and returns as its value the edge e^* such that $P(e^*) = \min_{e \in s}\{P(e)\}$. (As usual, if there is more than one such edge, then *Dijkstra-nextEdge*(*G*,*S*) selects the one determined by the default priority.)

Correctness of Dijkstra's Algorithm

- **Theorem 4.3.3.** Let T_j be the Dijkstra tree after j iterations of Dijkstra's algorithm on a connected graph G, for $0 \le j \le |V_G| - 1$. Then for each v in T_j, the unique s-v path in T_j is a shortest s-v path in G.
  - ✓ Prove by induction. Assume for some j, $0 \le j \le |V_G| - 2$, that T_j satisfies the property. T_{j+1} is the result of $(j+1)$-*th* iteration of Dijkstra algorithm by adding a frontier edge e and new vertex y (from x to y) to T_j.
  - ✓ We must prove for any s-y path R in G $length(R) \ge P(e)$, i.e., the s-y path Q in T_{j+1} is a shortet s-y path in G
  - ✓ Edge f (connect v and z) is the first edge to leave T_j , and vertex v is in T_j.
  - ✓ Algorithm selects e instead of f in $(j+1)$-*th* iteration to produce T_{j+1}, implying $P(f) \ge P(e)$. Thus

$$length(R) = dist[v] + w(f) + length(K) = P(f) + length(K) \ge P(e) + length(K) \ge P(e) = length(Q)$$



- **COMPUTATIONAL NOTE:** Each time a new vertex y is added to the Dijkstra tree, the set of frontier edges can be updated without recomputing any P-value. In particular, delete each previous frontier edge incident on vertex y (because both its endpoints are now in the tree), **and compute the priorities of the new frontier edges only (i.e., those that are incident on vertex y).** This modification significantly decreases the running time.

4.4 Application of Depth-First Search

The Finish Order of a Depth-First Search

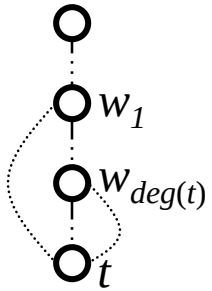
- **DEFINITION**: During a depth-first search, a discovered vertex *v* is ***finished*** when all of its neighbors have been discovered. (There is no frontier edge with tree endpoint *v*)
- **DEFINITION**: The ***finish order*** of a depth-first search is the order in which the vertices are finished.
- **Remark**: In §4.2, we observed that **a pre-order traversal of a dfs-tree reproduces the discovery order** (Exercise 4.2.17). Regarding finish order, observe that the root is the last vertex finished in a depth-first search. In fact, **a post-order traversal (§3.3) of the dfs-tree reproduces the finish order generated by the depth-first search** (see Exercises).

Growing a DFS-Path

- **DEFINITION**: A ***dfs-path*:** a dfs-path is the depth-first tree produced by executing a depth-first search and stopping before you backtrack for the first time.
- **Proposition 4.4.1.** *Let G be a graph, and let P be the dfs-path produced by executing a depth-first search until the first vertex t becomes finished. Then all of the neighbors of t lie on path P.*
  - ✓ Otherwise, *t* has at least one frontier edge and the search will not backtrack.

Growing a DFS-Path

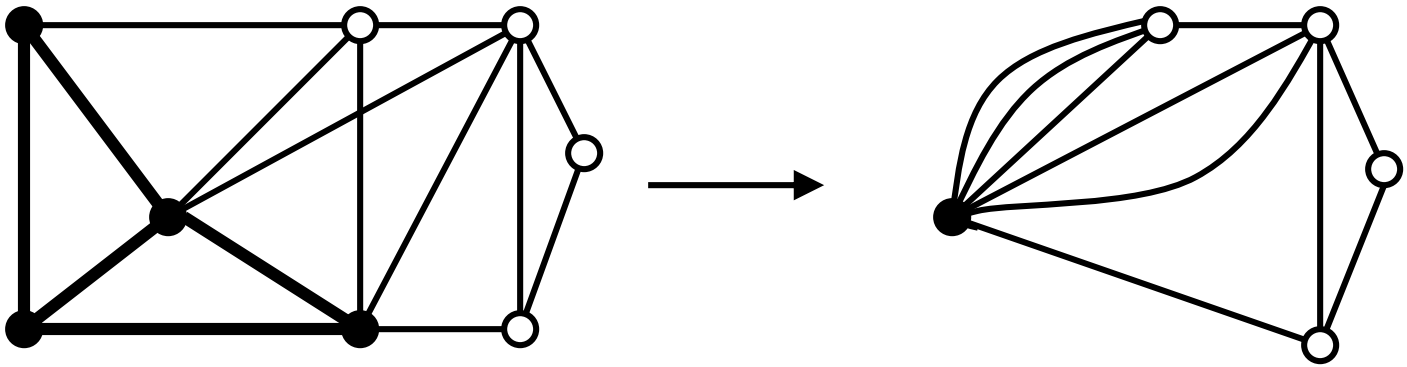
- **Corollary 4.4.2.** *Let G be a connected graph with at least two vertices. If a dfs-path is grown until the first vertex t is finished, then either deg(t) = 1 or t and all its neighbors lie on a cycle of G.*
  - ✓ By Proposition 4.4.1, all neighbors of *t* are on the dfs-path. Assume $deg(t) \geq 1$, w_1 is *t's* neighbor with least *dfnumber* and $w_{deg(t)}$ is *t's* neighbor with largest *dfnumber*. Then the dfs-path from w_1 to *t* plus edge tw_1 form a cycle.



- **Corollary 4.4.3.** *Let G be a connected simple graph with minimum degree δ. Then G contains a cycle of length greater than δ. ($\delta > 1$)*
  - ✓ By Corollary 4.4.2, *G* contains a cycle with at least $(\delta + 1)$ vertices and length $(\delta + 1)$

Finding the Cut-Edges of a Connected Graph

- **DEFINITION**: Let B be the set of cut-edges (bridges) of a connected graph G. A ***bridge component*** (***BC***) of G is a component of the subgraph $G - B$ (§2.4).
- **DEFINITION**: Let H be a subgraph of a graph G. The ***contraction of H to a vertex*** is the replacement of H by a single vertex k. Each edge that joined a vertex $v \in V_G - V_H$ to a vertex in H is replaced by an edge with endpoints v and k.
- **Example 4.4.1.**

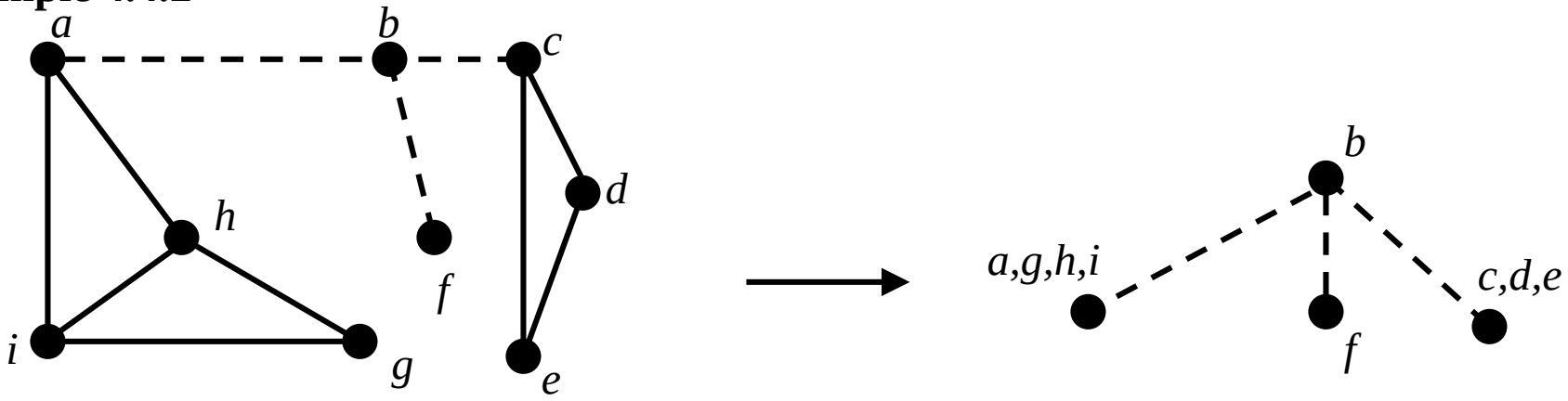


Contracting *H* to a vertex

- **Proposition 4.4.4.** *Let G be a connected graph. All vertices on a cycle are in the same bridge component of G.*
  - ✓ Every edge on a cycle is a cycle edge but not a cut edge, and then will not cross different bridge components. Crossing different bridge components has to pass a bridge edge.

Finding the Cut-Edges of a Connected Graph

- **Proposition 4.4.5.** *Let G be a connected graph. The graph that results from contracting each bridge component of G to a vertex is a tree.*
- **Example 4.4.2**



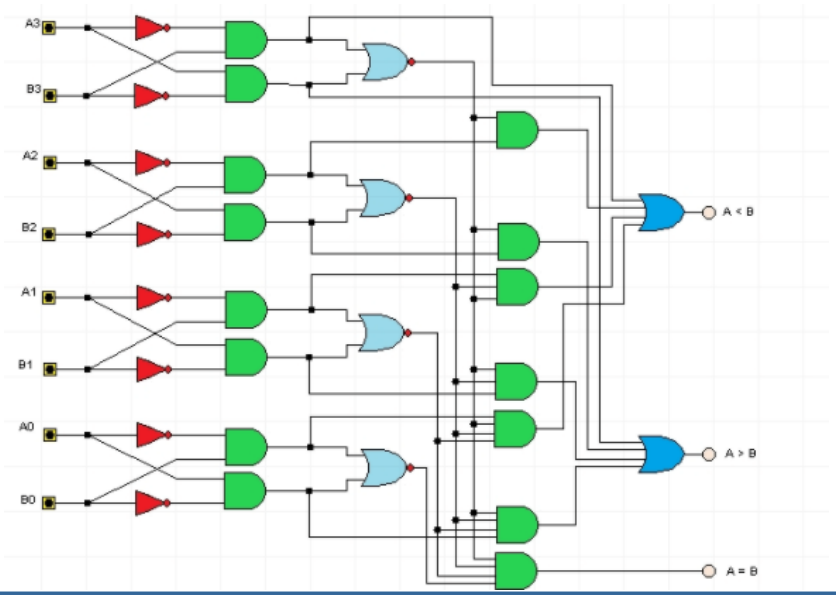
Contracting each bridge component to a vertex

- **Proposition 4.4.6.** *Let v be a vertex of a connected graph G with deg(v) ≤ 1. Then the bridge components of G are the trivial subgraph x and the bridge components of G – x.*
  - ✓ Trivial graph is a graph with only one vertex and no edge. Thus $x = \{v\}$.
- **Algorithm 4.4.1: Finding Cut-Edges**
  - ✓ Check the process with Example 4.4.2
- **COMPUTATINAL NOTE**: For efficiency, the dfs-path grown in each iteration should start at the same vertex and use as much of the previous dfs-path as possible.

DFS

Algorithm 4.4.1: Finding Cut-Edges
Input: a connected graph G
Output: the cut edges of G
Initialize graph H as graph G.
While $|V_H| > 1$
    Grow a dfs-path to the first vertex t tat becomes finished.
    If $deg(t) = 1$
        Mark the edge incident on t as a bridge.
        $H = H - 1$.
    else
        Let H be the result of contracting cycle C to a vertex.

Algorithm 4.4.2: Topological sort
Input: an n-vertex dag G
Output: a topological sort ts of G
Initialize graph H as graph G.
Initialize $k = n$.
While $|V_H| \neq 0$
    Start at a source and grow a dfs-path until the first vertex t is finished.
    Set $ts(t) = k$.
    $k = k - 1$.
    $H = H - t$.



Topological Sorting by Depth-first Search

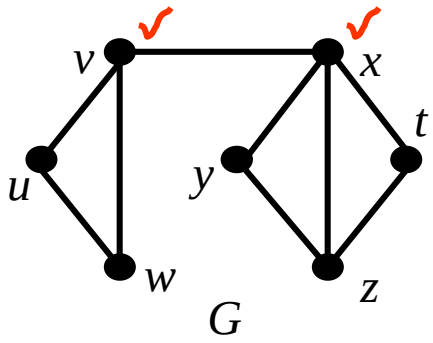
- **DEFINITION**: A ***dag*** is a directed acyclic graph, i.e., it has no directed cycles.
- **DEFINITION**: A ***topological sort*** (or ***topological order***) of an n-vertex dag G is a bijection ts from the set $\{1, 2, \ldots, n\}$ to the vertex-set V_G such that every arc e is directed from a smaller number to a larger one, i.e., $ts(tail(e)) < ts(head(e))$.
- **DEFINITION** : A ***source*** in a dag is a vertex with indegree 0, and a ***sink*** is a vertex with outdegree 0.
- **Proposition 4.4.7.** *Every dag has at least one source and at least one sink.*
  - ✓ If a dag has no source, then there must be a cycle
  - ✓ Similarly, a dag must have a cycle if it has no sink.
- **Remark**: Simply using the discovery order (i.e., the dfnumbers) produced by a depth-first search will not necessarily be a topological sort because right-to-left cross arcs may occur (Proposition 4.2.2). However, the reverse finish order can be used.
- **Proposition 4.4.8.** *Let G be a dag, and let t be the vertex that becomes finished during an execution of a depth-first search. Then t is a sink.*
  - ✓ If t were not a sink, then G would contain a directed cycle (since the vertex directed from t must be in the dfs-path).
- **Algorithm 4.4.2: Topological Sort**

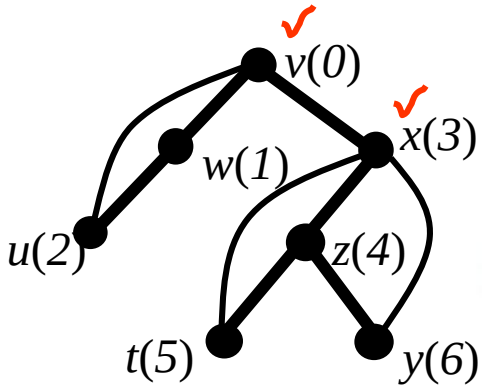
Topological Sorting by Depth-first Search

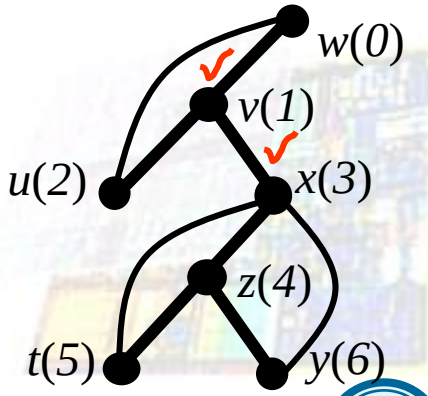
- **COMPUTATIONAL NOTE**: Again, as in Algorithm 4.4.1, efficiency dictates that the dfs-path grown in each iteration should start at the same vertex and use as much of the previous dfs-path as possible.

Finding the Cut-Vertices of a Connected Graph

- **Proposition 4.4.9.** *A vertex v in a connected graph is a cut-vertex iff there exist two distinct vertices u and w, both different from v, such that v is on every u-w path in the graph.*
  - ✓ Necessity (→) If *v* is a cut-vertex of a connected graph *G*, then there are vertices, say *u* and *w*, in separate components of *G-v*. It follows that every *u-w* path in *G* must contain *v*.
  - ✓ Sufficiency(←) If *u* and *v* are vertices of a connected graph *G*, such that every *u-w* path in *G* contains *v*, then *G-v* contains no *u-w* path. Thus *u* and *w* are in different components of *G-v*.
- **Example 4.4.3:**
  - ✓ two observations

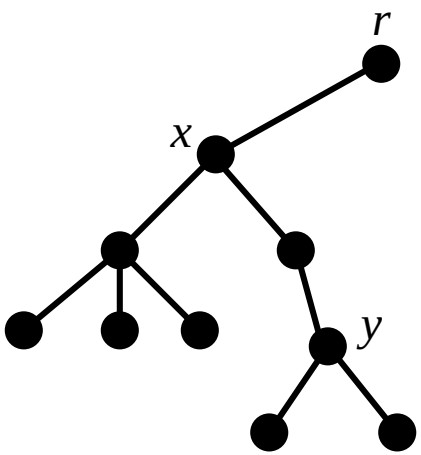


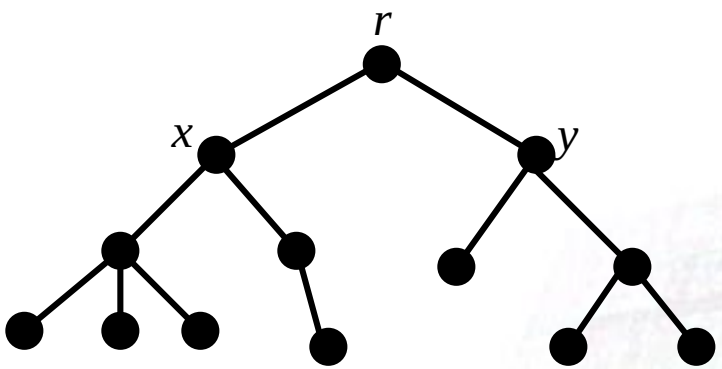




Finding the Cut-Vertices of a Connected Graph

- **Proposition 4.4.10.** *Let tree T be the result of applying depth-first search to a connected graph G. Then the root r of T is a cut-vertex of G iff r has more than one child in T.*
  - ✓ *Necessity* ($\rightarrow$) By way of contrapositive, assume r has only one child x in T, then $T - r$ is still a tree with root x. Then r is not a cut vertex.
  - ✓ *Sufficiency* ($\leftarrow$) Assume r has two children, x and y. By Proposition 4.2.1, there is no edge in G (cross edge in T) connecting any vertex in the subtree rooted at x with any one vertex in the subtree rooted at y. Thus, $T - r$ has two isolated subtrees rooted at x and y, implying r is a cut vertex of G.

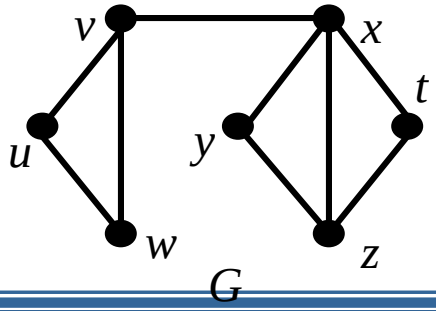


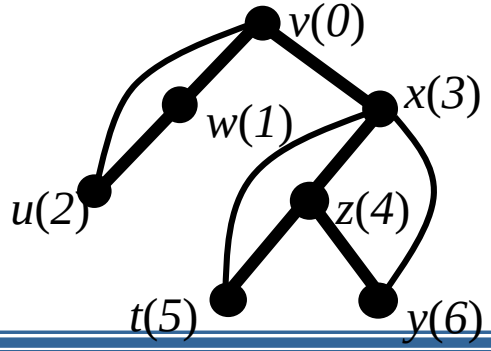


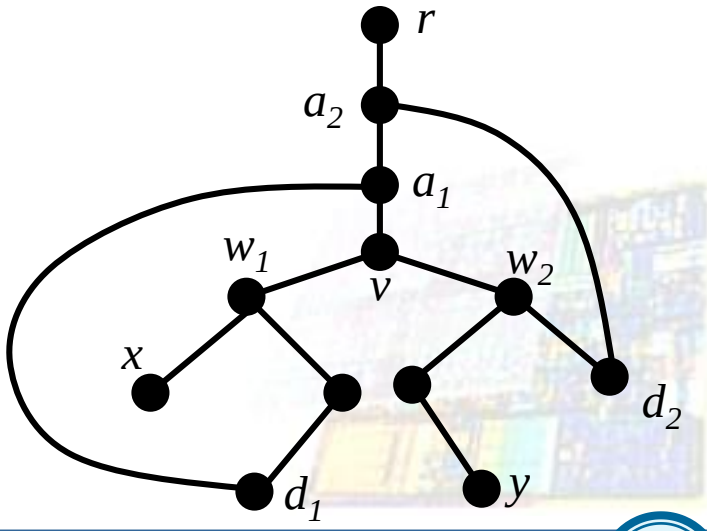
Finding the Cut-Vertices of a Connected Graph

- **Proposition 4.4.11.** *Let tree T be the result of applying depth-first search to a connected graph G. Then a non-root v of T is a cut-vertex of G if and only if* ***v has a child w*** *such that no descendant of w is joined to a proper ancestor of v by a non-tree edge.*

  - ✓ *Necessity* ($\rightarrow$)
    - By way of contrapositive, suppose that every child of v has a descendant joined to a proper ancestor of v. First suppose that x and y are any two proper descendants of v. We claim that there exists an x-y path that does not contain v. (v is not a cut vertex).
    - Case 1: x and y are descendants of the same child of v, then the claim is trivially true.
    - Case 2: x and y are descendants of two different children of v, say w_1 and w_2, respectively. Vertex w_1 (w_2) has a descendant d_1 (d_2) joined by a non-tree edge to a proper ancestor of v, say a_1 (a_2).
  - ✓ *Sufficiency* ($\leftarrow$) Then every path in G from w to r must pass through v, and, hence, v is a cut vertex.

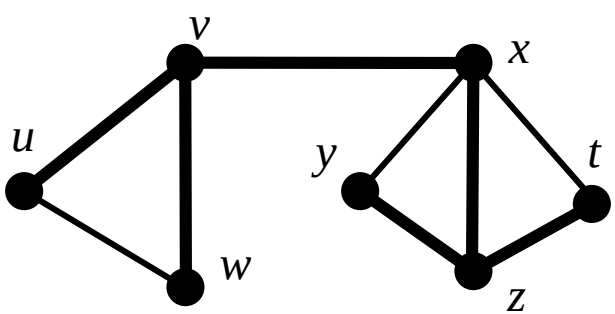


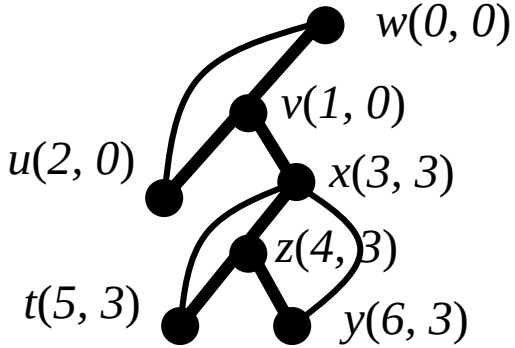




Characterizing Cut-Vertex in Terms of the *dfnumbers*

- Propositions 4.2.1 (dfs-trees have no cross edges) and 4.4.11→a non-root *v* of a depth-first search tree is a cut-vertex if and only if *v* has a child *w* such that no descendant of *w* is joined by a non-tree edge to a vertex whose *dfnumber* is smaller than *dfnumber*(*v*).
- **NOTATION** : Let *low*(*w*) denote the smaller of *dfnumber*(*w*) and the smallest *dfnumber* among all vertices joined by a non-tree edge to some descendant of *w*.
- **Corollary 4.4.12.** *Let tree T be the result of applying depth-first search to a connected graph. Then a non-root v of T is a cut-vertex if and only if* ***v has a child w*** *such that low*(*w*) ≥ *dfnumber*(*v*).
- **Algorithm 4.4.3.**
- **Example 4.4.4:**





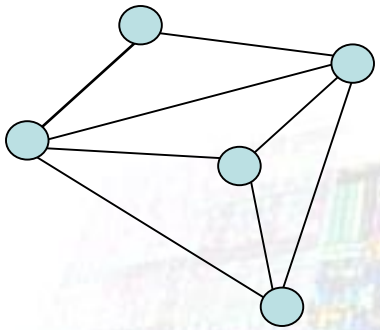
- **COMPUTATIONAL NOTE**: It is not hard to show that for any vertex *v*, *low*(*v*) is the minimum of the following values:
  - ✓ a. *dfnumber*(*v*),
  - ✓ b. *dfnumber*(*z*) for any *z* joined to *v* by a non-tree edge (*u*, *t*, *y*), and
  - ✓ c. *low*(*y*) for any child *y* of *v*. (*v*, *z*)

Finding Cut-Vertices

Algorithm 4.4.3: Finding Cut Vertices
Input: a connected graph G
Output: a set K of the cut vertices of graph G
Initialize the set K of cut vertices as empty.
Choose an arbitrary vertex r of graph G.
Execute a dfs of graph G, starting at vertex r.
Let T be the output tree.
If root r has more than one child in T
    Add vertex r to set K.
For each vertex w
    Compute $low(w)$.
For each non-root v
    If there is a child w of v such that $low(w) \geq dfnumber(v)$
        Add vertex v to set K.

Escaping From a Maze: Tarry's Algorithm

- This relationship allows the computation of the *low* values to take place *during* the depth-first search. In particular, when the vertex *v* has been backtracked to for the last time, the *low* values of each of the children of *v* will have already been computed, allowing *low*(*v*) then to be computed. (For further details, see [AhHoUl83].)
- *Problem*: You are inside a room in a maze consisting of a network of tunnels and rooms. At the end of each tunnel is an open doorway through which you can enter or exit a room. Your goal is to find your way to a freedom room from which you can exit the maze, if such a room exists, or determine that no such room exists. Each room has a piece of marking chalk and nothing else.
  - ✓ 1. choose one unmark, go out and mark OUT; 2. enter a room the first time (no mark), mark IN; 3. Choose one door marked IN to go out as the room's all doors are marked; 4. enter a room with all doorways marked out, stop.



- ***Tarry's algorithm*:**
  - ✓ Never backtrack through a tunnel unless there is no alternative
  - ✓ Never go through a tunnel a second time in the same direction
- **Proposition 4.4.13.** *If there is no freedom room in the maze, Tarry's algorithm will stop after each tunnel is traversed exactly twice, once in each direction.*

Escaping From a Maze: Tarry's Algorithm

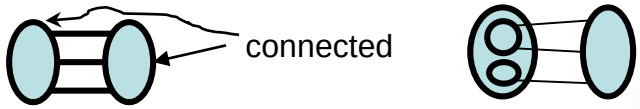
- If you are in a room that is not the freedom room, and it has at least one unmarked doorway, then pass through one of the unmarked doorways to a tunnel, mark that doorway OUT, traverse the tunnel, pass through the doorway at the other end, and enter the room there.
- If you pass through a doorway and enter a room that is not the freedom room, and it has all doorways unmarked, mark the doorway you pass through IN.
- If you pass through a doorway and enter a room that is not the freedom room, and it has all doorways marked, pass through a doorway marked IN, if one exists, traverse the tunnel, pass through the doorway at the other end, and enter the room there.
- If you pass through a doorway and enter the room with all doorways marked OUT, stop.

4.5 Cycles, Edge-Cuts, Spanning Trees

- **Proposition 4.5.1.** *A graph G is connected if and only if it has a spanning tree.*
  - ✓ *Necessity* ($\rightarrow$)
    - Assume T is the connected spanning subgraph of G with the least number of edges and T has cycles.
    - Then we can remove an edge in a cycle from T such that T is still a connected spanning subgraph. Contradiction to the edge minimality of T. Thus T has no cycle.
  - ✓ *Sufficiency* ($\leftarrow$) Spanning tree is connected, so G is connected.
- **Proposition 4.5.2.** *A subgraph H of a connected graph G is a subgraph of some spanning tree if and only if H is acyclic.*
  - ✓ *Necessity* ($\rightarrow$) H is a spanning tree's subgraph, then H is acyclic by definition.
  - ✓ *Sufficiency* ($\leftarrow$)
    - Let H be an acyclic subgraph of G, and let T be any spanning tree of G.
    - Consider the connected, spanning subgraph G_1, where $V_{G1} = V_T \cup V_H$ and $E_{G1} = E_T \cup E_H$.
    - If G_1 is acyclic, then H is contained in T.
    - Otherwise suppose G_1 has a cycle C_1. Since H is acyclic, there is an edge e_1 in C_1 but not in H.
    - We can remove e_1 to get G_2 that is still a connected spanning subgraph of G and still contains H. If G_2 is acyclic, then we get a spanning tree, or repeat the same process to get an acyclic spanning tree of G.

Partition-Cuts and Minimal Edge-Cuts

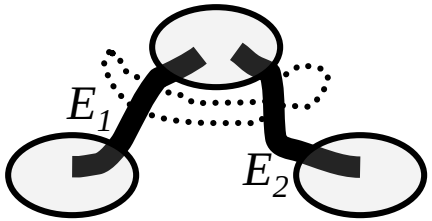
- **Review *edge-cut***
- **DEFINITION**: Let G be a graph, and let X_1 and X_2 form a partition of V_G. The set of all edges of G having one endpoint in X_1 and the other endpoint in X_2 is called a ***partition-cut*** of G and is denoted $\langle X_1, X_2 \rangle$.
- **Proposition 4.5.3.** *Let $\langle X_1, X_2 \rangle$ be a partition-cut of a connected graph G. If the subgraphs of G induced by the vertex sets X_1 and X_2 are connected , then $\langle X_1, X_2 \rangle$ is minimal edge-cut.*
  - ✓ We have to prove (1) $\langle X_1, X_2 \rangle$ is an edge cut and (2) any proper subset of $\langle X_1, X_2 \rangle$ is not an edge cut, then $\langle X_1, X_2 \rangle$ is a minimal edge cut.
  - ✓ (2) Assume S is any proper subset of $\langle X_1, X_2 \rangle$, then there is at least one edge $e \in \langle X_1, X_2 \rangle - S$ (e connects two connected subgraphs of $G - \langle X_1, X_2 \rangle$).
  - ✓ Thus $G - S$ is connected, and any proper subset S of $\langle X_1, X_2 \rangle$ is not a edge cut, implying $\langle X_1, X_2 \rangle$ is a minimal edge cut.



- **Proposition 4.5.4.** *Let S be a minimal edge-cut of a connected graph G, and let X_1 and X_2 be the vertex-sets of the **two components of $G - S$**. Then $S = \langle X_1, X_2 \rangle$.*
  - ✓ S is minimal, so $S \subset \langle X_1, X_2 \rangle$. If there exists an edge $e \in \langle X_1, X_2 \rangle - S$, then its endpoints would lie in the same component of $G - S$, contradicting the definition of $<X_1, X_2>$.

Partition-Cuts and Minimal Edge-Cuts

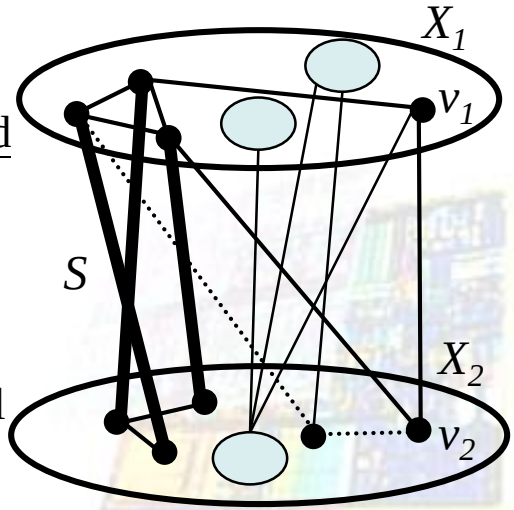
- **Remark**: The premise of Proposition 4.5.4 assumes that the removal of a minimal edge-cut from a connected graph creates *exactly* two components. This is a generalization of Corollary 2.4.3 and can be argued similarly, using the minimality condition (see Exercises 4.5.18).



✓$E_1 \cup E_2$ is a partition cut as well as an edge cut, but not a minimal edge cut.
✓E_1 and E_2 are both minimal edge cuts and partition cuts.

- **Proposition 4.5.5.** *A partition-cut $\langle X_1, X_2\rangle$ in a connected graph G is a minimal edge-cut of G or union of edge-disjoint minimal edge-cuts.*

✓ Since $\langle X_1, X_2\rangle$ is an edge cut of G, it must contain a minimal edge cut, say S. If $\langle X_1, X_2\rangle \neq S$, let $e \in \langle X_1, X_2\rangle - S$, where the endpoints v_1 and v_2 of e lie in X_1 and X_2, respectively. Since S is a minimal edge cut, the X_1-endpoints of S are in one component of $G - S$, say S_1, and the X_2-endpoints of S are in the other component, say S_2. Furthermore, v_1 and v_2 are in the same component of $G - S$ ($e \notin S$). Suppose without loss of generality, v_1 and v_2 are in S_1, then every path in G from v_1 to v_2 must use at least one edge of $\langle X_1, X_2\rangle - S$. Thus $\langle X_1, X_2\rangle - S$ is an edge cut of G and, hence, contains a minimal edge cut R. Repeat to check if $\langle X_1, X_2\rangle - (S \cup R)$ is empty. Eventually, we get $\langle X_1, X_2\rangle - (S_1 \cup S_2 \cup \ldots \cup S_r) = \phi$.

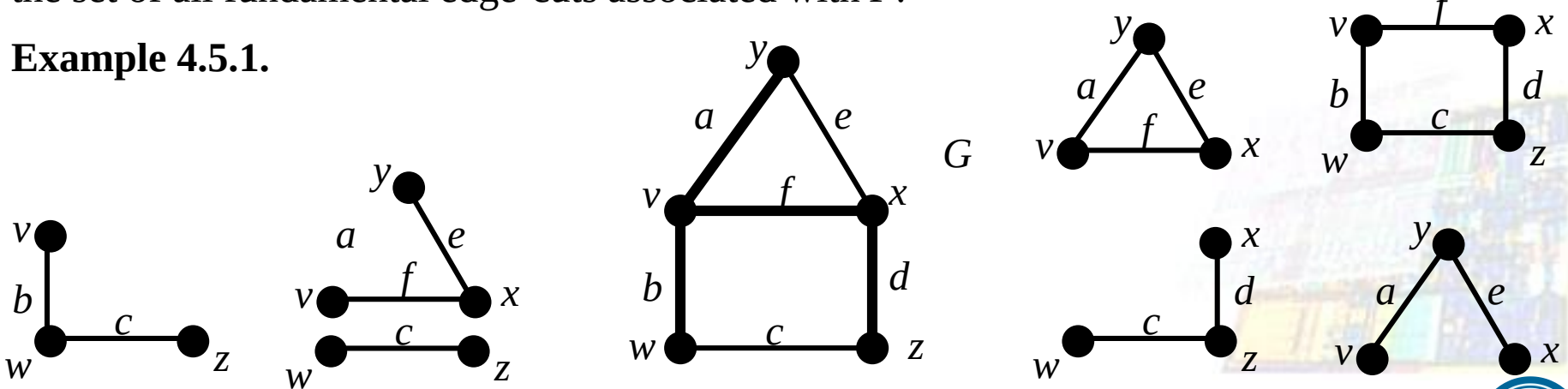


Fundamental Cycles and Fundamental Edge-Cuts

- DEFINITION: Let G be a graph with $c(G)$ components. The ***edge-cut rank*** of G is the number of edges in a full spanning forest of G. Thus, by Corollary 3.1.4, the edge-cut rank equals $|V_G| - c(G)$. (You can regard this as the number of edge cuts)
- **Review *full spanning forest* and *relative complement of H* (in *G*)**
- DEFINITION: Let G be a graph with $c(G)$ components. The ***cycle rank*** (or ***Betti number***) of G, denoted $\beta(G)$, is the number of edges in the relative complement of a full spanning forest of G. Thus, the cycle rank is $\beta(G) = |E_G| - |V_G| + c(G)$. (You can regard this as the number of cycles or edge redundancies)
- **Remark**: Observe that *all* of the edges in the relative complement of a spanning forest could be removed without increasing the number of components. Thus, the cycle rank $\beta(G)$ equals the maximum number of edges that can be removed from G without increasing the number of components. Therefore, $\beta(G)$ is a measure of the *edge redundancy* with respect to the graph's connectedness.
- DEFINITION: Let F be a full spanning forest of a graph G, and let e be any edge in the relative complement of forest F. The cycle in the subgraph $F + e$ (existence and uniqueness guaranteed by the characterization theorem in §3.1) is called a ***fundamental cycle of*** G ***(associated with the spanning forest*** F***)***.
  - ✓ Fundamental cycle: one non-tree edge and the other are tree edges.

Fundamental Cycles and Fundamental Edge-Cuts

- **Remark**: Each of the edges in the relative complement of a full spanning forest F gives rise to a *different* fundamental cycle.
- DEFINITION: The ***fundamental system of cycles*** associated with a full spanning forest F of a graph G is the set of all fundamental cycles of G associated with F.
- DEFINITION: Let F be a full spanning forest of a graph G, and let e be any edge of F. Let V_1 and V_2 be the vertex-sets of the two new components of the edge-deletion subgraph $F - e$. Then the partition-cut $\langle V_1, V_2 \rangle$, which is a minimal edge-cut of G by Proposition 4.5.3, is called a ***fundamental edge-cut*** (***associated with*** F). (one tree edge + other non-tree edges)
- **Remark**: For each edge of F, its deletion gives rise to a different fundamental edge-cut.
- DEFINITION: The ***fundamental system of edge-cuts*** associated with a full spanning forest F is the set of all fundamental edge-cuts associated with F.
- **Example 4.5.1.**



Relationship Between Cycles and Edge-Cuts

- **Proposition 4.5.6.** *Let S be a set of edges in a connected graph G. Then S is an edge-cut of G if and only if every spanning tree of G has at least one edge in common with S.*
  - ✓ By Proposition 4.5.1, S is an edge cut if and only if $G - S$ contains no spanning tree of G, implying $\forall\ T_i$ of G, $S \cap T_i \neq \phi$.
- **Proposition 4.5.7.** *Let C be a set of edges in a connected graph G. Then C contains a cycle if and only if the relative complement of every spanning tree of G has at least one edge in common with C.*
  - ✓ By Proposition 4.5.2, edge set C contains a cycle if and only if C is not contained in any spanning tree of G, which means that the relative complement of every spanning tree of G has at least one edge in common with C.
- **Proposition 4.5.8.** *A cycle and a minimal edge-cut of a connected graph have an even number of edges in common.*
  - ✓ A minimal edge cut partitions a vertex set into two subsets. If the vertices of a cycle are in the same subset, then they have no edge in common.
  - ✓ If the vertices of a cycle are in two subsets, as the cycle enters the other subset from one subset, it must return to the original subset. Thus the cycle must use even edges in edge cut to cross two subsets.

Relationship Between Cycles and Edge-Cuts

- **Example 4.5.1.** check the number of edges of three cycles in common with each minimal edge-cut in Fig. 4.5.2.
- **Proposition 4.5.9.** *Let T be a spanning tree of a connected graph, and let C be a fundamental cycle with respect to an edge e^* in the relative complement of T. Then the edge-set of cycle C consists of edge e^* and those edges of tree T whose fundamental edge-cuts contain e^*.*
  - ✓ Let $e_1, \ldots, e_k$ be the edges of T that, with e^*, make up the cycle C, and let S_i, be the fundamental edge-cut with respect to e_i, $1 \le i \le k$. We'll first prove that every S_i contains e^*. And then prove every S_j does not contain e^*, for $e_j \notin \{e_1, \ldots, e_k\}$.
  - ✓ Edge e_i is **the only edge of T** common to both C and S_i (by the definitions of C and S_i).
  - ✓ By Proposition 4.5.8, C and S_i must contain an even number of edges, and, hence, there must be an edge in the relative complement of T that is also common to both C and S_i. But e^* is the only edge in the complement of T that is in C. Thus, the fundamental edge-cut S_i must contain e^*, $1 \le i \le k$.
    - ➢ e_i and e^* are two common edges to both C and S_i.
  - ✓ To complete the proof, we must show that no other fundamental edge-cuts associated with T contain e^*. So let S be the fundamental edge-cut with respect to some edge b of T, different from $e_1, \ldots, e_k$. Then S does not contain any of the edges $e_1, \ldots, e_k$ (by S's definition). The only other edge of cycle C is e^*, so by Proposition 4.5.8, edge-cut S cannot contain e^*.

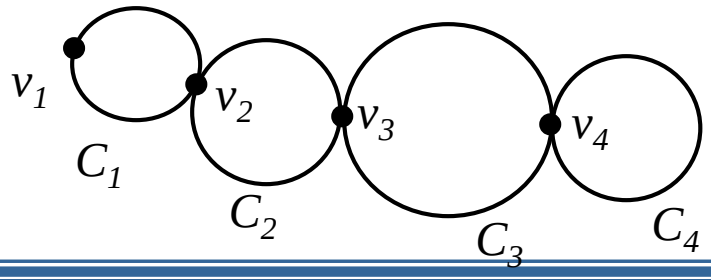
Relationship Between Cycles and Edge-Cuts

- **Example 4.5.1.** Verify Propositions 4.5.9 and 4.5.10 with cycle *a–e–f* any edge cut
- **Proposition 4.5.10.** *The fundamental edge-cuts with respect to an edge e of a spanning tree T consists of e and exactly those edges in the relative complement of T whose fundamental cycles contain e.*
- **Theorem 4.5.11 [Eulerian-Graph Characterization].** *The following statements are equivalent for a connected graph G.*

  1. G is eulerian. 2. The degree of every vertex in G is even.

  3. E_G is the union of the edge-sets of a set of edge-disjoint cycles of G.

  - ✓ (1→2) every pass-in to a vertex must follow a go-out to the vertex.
  - ✓ (2 →3) G is conneected and each vertex has even degree, so G must not be a tree. G has a cycle, say C_1. If $G_1 = C_1$, proof is complete. Otherwise let $G_1 = G - E_{C1}$. Since the degree of every vertex in C_1 is decreased by two, so every vertex in G_1 also has even degree. Thus G_1 also has a cycle…
  - ✓ (3→1)



4.6 Graphs and Vector Spaces

Vector Space of Edge Subsets

- **NOTATION**: For a graph G, let $W_E(G)$ denote the set of all subsets of E_G.
- **DEFINITION**: The ***ring sum*** of two elements of $W_E(G)$, say E_1 and E_2, is defined by
  - ✓ $E_1 \oplus E_2 = (E_1 - E_2) \cup (E_2 - E_1) = (E_1 \cup E_2) - (E_1 \cap E_2)$
- **Proposition 4.6.1.** *$W_E(G)$ is a vector space over $GF(2)$.*
- **TERMINOLOGY**: The vector space $W_E(G)$ is called the ***edge space of*** G.
- **Proposition 4.6.2.** *Let $E_G = \{e_1, e_2, \ldots, e_m\}$ be the edge-set of a graph G. Then the subsets $\{e_1\}, \{e_2\}, \ldots, \{e_m\}$ form a basis for the edge space $W_E(G)$. Thus, $W_E(G)$ is an m-dimensional vector space over $GF(2)$.*
  - ✓ If $H = \{e_{i_1}, e_{i_2}, \ldots, e_{i_r}\}$ is any vector in $W_E(G)$, then $H = \{e_{i_1}\} \oplus \{e_{i_2}\} \oplus \ldots \oplus \{e_{i_r}\}$. Clearly, the elements $\{e_{i_1}\}, \{e_{i_2}\}, \ldots, \{e_{i_r}\}$ are also linearly independent.
- **DEFINITION**: Let $s_1, s_2, \ldots, s_n$ be any sequence of objects, and let A be a subset of $S = \{s_1, s_2, \ldots, s_n\}$. The ***characteristic vector*** of the subset A is the n-tuple whose jth component is 1 if $s_j \in A$ and 0 otherwise.
- **Example.** $S = \{e_1, e_2, e_3, e_4, e_5\}$, $A = \{e_2, e_4, e_5\}$, characteristic vector of A = (0,1,0,1,1)

Vector Space of Edge Subsets

- A general result from linear algebra states that every m-dimensional vector space over a given field F is isomorphic to the vector space of m-tuples over F.
- For the vector space $W_E(G)$, this result may be realized as:
  - ✓ If $E_G = \{e_1, e_2, \ldots, e_m\}$, then the mapping *charvec* that assigns each subset of E_G to its *characteristic vector* is an isomorphism from $W_E(G)$ to the vector space of m-tuples over GF(2).
  - ✓ If E_1 and E_2 are two subsets of E_G, then the definitions of the ring-sum operator $\oplus$ and mod 2 component-wise addition $+_2$ imply

$$charvec(E_1 \oplus E_2) = charvec(E_1) +_2 charvec(E_2)$$

  - ✓ **Example.** $charvec(\{e_1, e_2, e_3\} \oplus \{e_2, e_5\}) = charvec(\{e_1, e_3, e_5\}) = (1,0,1,0,1)$

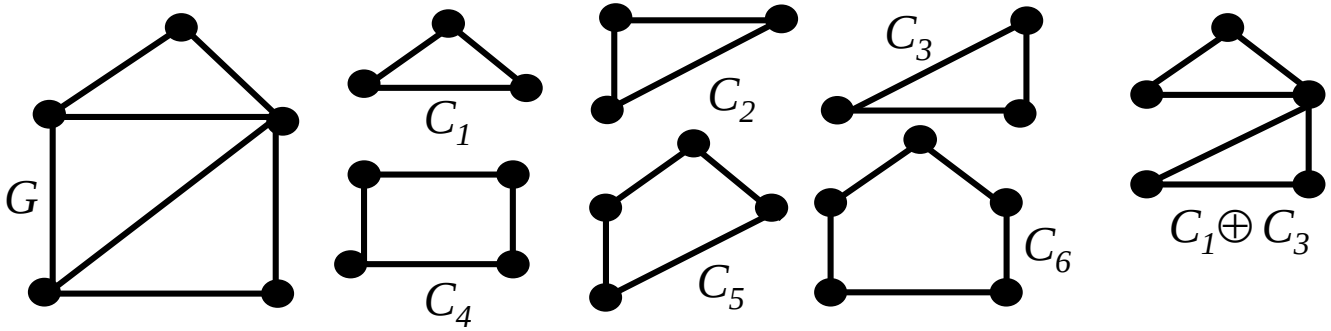
$$(1,1,1,0,0) +_2 (0,1,0,0,1) = (1,0,1,0,1)$$

- **Remark**: Each subset E_i of E_G uniquely determines a subgraph of G, namely, the edge-induced subraph $G_i = G(E_i)$. Thus, the vectors of $W_E(G)$ may be viewed as the **edge-induced subgraphs** G_i instead of as the edge subsets E_i. Accordingly, $1 * G_i = G_i$ and $0 * G_i = \varnothing$, where G_i is any edge-induced subgraph of G and where $\varnothing$ now refers to the *null graph* (with no vertices and no edges). This vector space will still be denoted $W_E(G)$.

The Cycle Space of a Graph-

- **DEFINITION**: The ***cycle space*** of a graph G, denoted $W_C(G)$, is the subset of the edge space $W_E(G)$ consisting of the null set (graph) $\varnothing$, all cycles in G, and all unions of edge-disjoint cycles of G.

- **Example 4.6.1.**

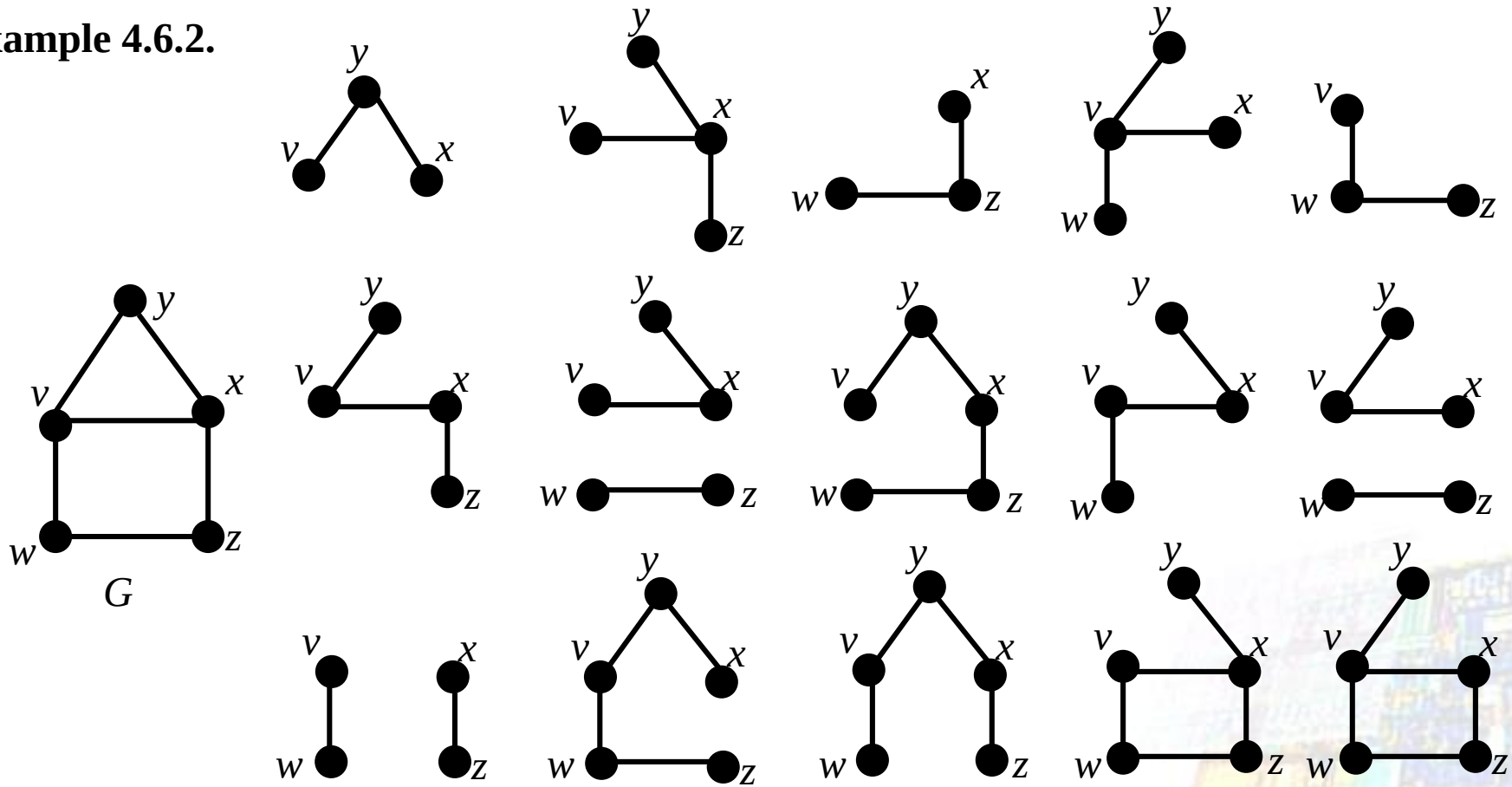


  - ✓ Each vector of $W_C(G)$ is a subgraph having no vertices of odd degrees.
  - ✓ the sum of any two of the vectors of $W_C(G)$ is again a vector of $W_C(G)$.

- **Proposition 4.6.3.** *Given a graph G, the cycle space $W_C(G)$ is a subspace of the edge space $W_E(G)$.*
  - ✓ It suffices to show that the elements of $W_C(G)$ are closed under $\oplus$. So consider any two distinct members C_1 and C_2 of $W_C(G)$ and let $C_3 = C_1 \oplus C_2$. By Theorem 4.5.11, it must be shown that $deg_{C_3}(v)$ is even for each vertex v in C_3. Consider any vertex v in C_3, and let X_i denote the set of edges incident with v in C_i for $i = 1, 2, 3$. Since $|X_i|$ is the degree of v in C_i, $|X_1|$ and $|X_2|$ are both even, and $|X_3|$ is nonzero. Since $C_3 = C_1 \oplus C_2$, $X_3 = X_1 \oplus X_2$. But this implies that $|X_3| = |X_1| + |X_2| - 2|X_1 \cap X_2|$, which shows that $|X_3|$ must be even.

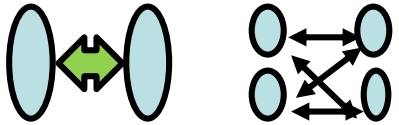
The Edge-Cut Subspace of a Graph

- **DEFINITION**: The ***edge-cut space*** of a graph G, denoted $W_S(G)$, is the subset of the edge space $W_E(G)$ consisting of the null graph $\varnothing$, all minimal edge-cuts in G, and all unions of edge-disjoint minimal edge-cuts of G.
- **Example 4.6.2.**



The Edge-Cut Subspace of a Graph

- **Proposition 4.6.4.** *Given a graph G, the edge-cut space $W_S(G)$ is closed under the ring-sum operation $\oplus$ and, hence, is a subspace of the edge space $W_E(G)$.*
  - ✓ By Proposition 4.5.3., it suffices to show that the ring sum of any two partition-cuts in a graph G is also a partition-cut in G. So let $S_1 = \langle X_1, X_2\rangle$ and $S_2 = \langle X_3, X_4\rangle$ be any two partition-cuts in G, and let $V_{ij} = X_i \cap X_j$, for $i = 1, 2$ and $j = 3, 4$.



  - ✓ Then the $\boldsymbol{V_{ij}}$ **are mutually disjoint**, with

    $S_1 = \langle V_{13} \cup V_{14}, V_{23} \cup V_{24}\rangle = \langle V_{13}, V_{23}\rangle \cup \langle V_{13}, V_{24}\rangle \cup \langle V_{14}, V_{23}\rangle \cup \langle V_{14}, V_{24}\rangle$

    and $S_2 = \langle V_{13} \cup V_{23}, V_{14} \cup V_{24}\rangle = \langle V_{13}, V_{14}\rangle \cup \langle V_{13}, V_{24}\rangle \cup \langle V_{23}, V_{14}\rangle \cup \langle V_{23}, V_{24}\rangle$
  - ✓ Hence, $S_1 \oplus S_2 = \langle V_{13}, V_{23}\rangle \cup \langle V_{14}, V_{24}\rangle \cup \langle V_{13}, V_{14}\rangle \cup \langle V_{23}, V_{24}\rangle$
  - ✓ But $\langle V_{13}, V_{23}\rangle \cup \langle V_{14}, V_{24}\rangle \cup \langle V_{13}, V_{14}\rangle \cup \langle V_{23}, V_{24}\rangle = \langle V_{13} \cup V_{24}, V_{14} \cup V_{23}\rangle$

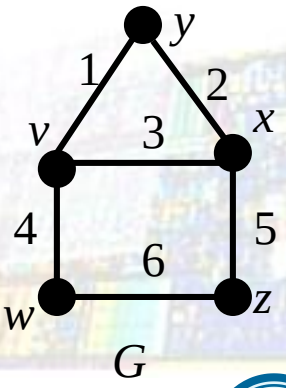
    which is a partition-cut in G, and the proof is complete.

- **Example 4.6.5.** $S_1 = \langle\{v,w\},\{x,y,z\}\rangle = \{1,3,6\}$, $S_2 = \langle\{v,x,y\},\{w,z\}\rangle = \{4,5\}$

  and $S = S_1 \oplus S_2 = \langle\{x,y,w\},\{v,z\}\rangle = \{1,3,5,4,6\}$

  $X_1 = \{v, w\}$, $X_2 = \{x, y, z\}$, $X_3 = \{v, x, y\}$, $X_4 = \{w, z\}$, $V_{13} = \{v\}$, $V_{14} = \{w\}$

  $V_{23} = \{x, y\}$, $V_{24} = \{z\}$, $V_{13} \cup V_{24}$,$=\{v, z\}$, $V_{14} \cup V_{23} = \{w, x, y\}$

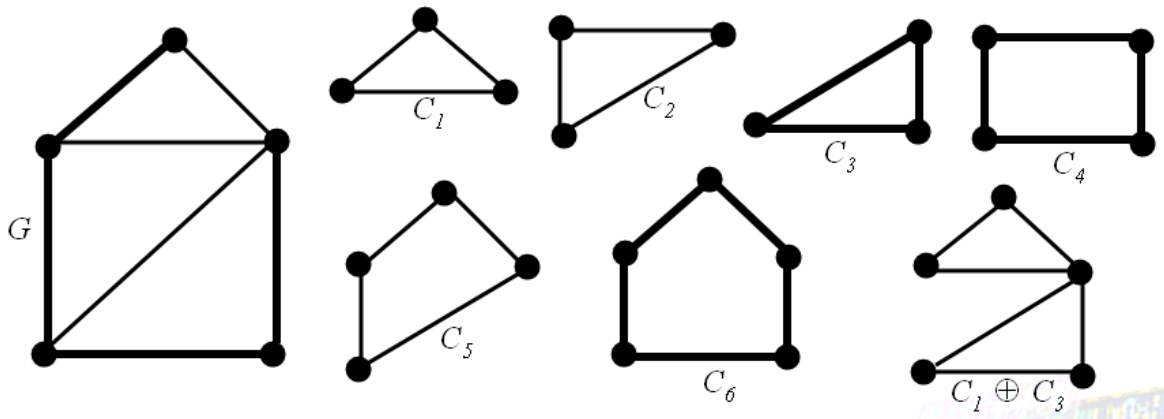


Bases for the Cycle and Edge-Cut Spaces

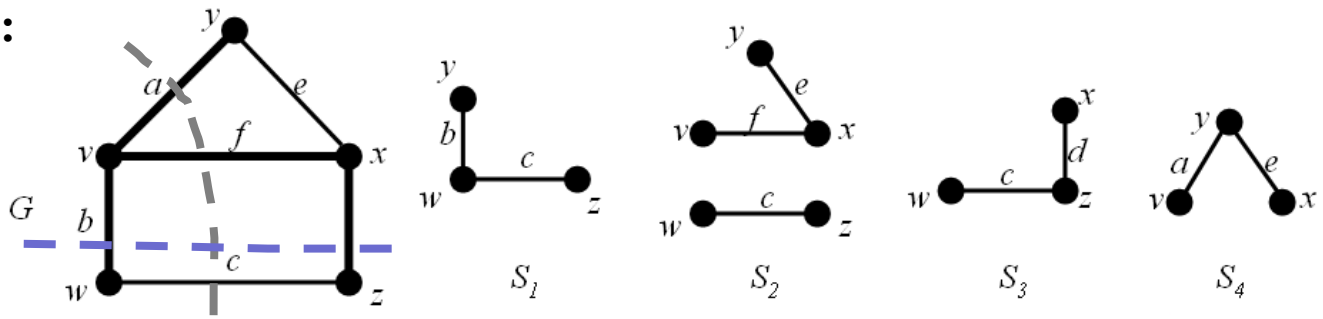
- **Theorem 4.6.5.** *Let T be a spanning tree of a connected graph G. Then the fundamental system of cycles associated with T is a basis for the cycle space $W_C(G)$.*
  - ✓ By the construction, each fundamental cycle associated with T contains exactly one non-tree edge that is not part of any other fundamental cycle associated with T. Thus, no fundamental cycle can be expressed as a ring sum of some or all of the other fundamental cycles. Hence, the fundamental system of cycles is a linearly independent set. To show the fundamental system of cycles spans $W_C(G)$, suppose H is any element of $W_C(G)$.
  - ✓ Now let $e_1, e_2, \ldots, e_r$ be the non-tree edges of H, and let C_i be the fundamental cycle in $T + e_i$, $i = 1, \ldots, r$. The completion of the proof requires showing that $H = C_1 \oplus C_2 \oplus \ldots \oplus C_r$, or equivalently, that $B = H \oplus C_1 \oplus C_2 \oplus \ldots \oplus C_r$ is the null graph.
  - ✓ Since each e_i appears only in C_i, and e_i is the only non-tree edge in C_i, B contains no non-tree edges. Thus, B is a subgraph of T and is therefore acyclic. But B is an element of $W_C(G)$ (since $W_C(G)$ is closed under ring sum), and the only element of $W_C(G)$ that is acyclic is the null graph.

Bases for the Cycle and Edge-Cut Spaces

- **Example 4.6.1 continued:**
  - ✓ $C_1 = C_4 \oplus C_6$
  - ✓ $C_2 = C_3 \oplus C_4$
  - ✓ $C_5 = C_3 \oplus C_6$
  - ✓ $C_1 \oplus C_3 = C_3 \oplus C_4 \oplus C_6$



- **Theorem 4.6.6.** *Let T be a spanning tree of a connected graph G. Then the fundamental system of edge-cuts associated with T is a basis for the edge-cut space $W_S(G)$.*
- **Example 4.6.2 continued:**

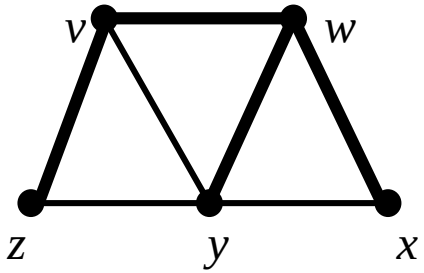


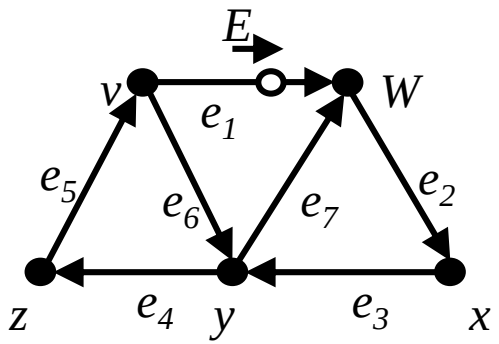
Bases for the Cycle and Edge-Cut Spaces

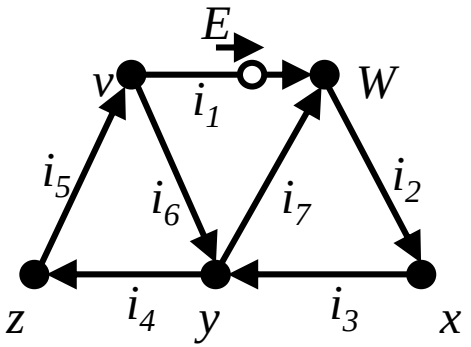
- **Application 4.6.1** *Applying Ohm's and Kirchhoff's Laws*:

$$
\begin{aligned}
&-i_1+i_5-i_6=0\\
&i_2-i_3=0\\
&i_4-i_5=0\\
&i_3-i_4+i_6-i_7=0\\
&i_1-i_2+i_7=0\\
&\hline\\
&0=0
\end{aligned}
$$

$$
\begin{aligned}
&i_1R_1-i_7R_7-i_6R_6-E=0 \quad (1)\\
&i_2R_2+i_3R_3+i_7R_7=0 \quad (2)\\
&i_1R_1+i_2R_2+i_3R_3-i_6R_6-E=0 \quad (3)\\
&\\
&(3) = (1) + (2)
\end{aligned}
$$







- A large network is likely to have a huge number of these redundancies.
- Since a fundamental system of cycles is a basis for the cycle space, their corresponding equations will constitute a full set of linearly independent equations.

$$
\begin{aligned}
&\langle z, v, w, y, z\rangle: \quad i_5R_5+i_1R_1-i_7R_7+i_4R_4-E=0\\
&\langle v, w, y, v\rangle: \quad i_1R_1-i_7R_7-i_6R_6-E=0\\
&\langle w, x, y, w\rangle: \quad i_2R_2+i_3R_3+i_7R_7=0
\end{aligned}
$$

Dimension of the Cycle Space and of the Edge-Cut Space

- **Corollary 4.6.7.** *Let G be a graph with $c(G)$ components. Then the dimension of the cycle space $W_C(G)$ is the cycle rank $\beta(G) = |E_G| - |V_G| + c(G)$, and the dimension of the edge-cut space $W_S(G)$ is the edge-cut rank $|V_G| - c(G)$.*
  - ✓ This follows directly from Theorems 4.6.5 and 4.6.6 and the definitions of cycle rank and edge-cut rank.

Relationship Between the Cycle and Edge-cut Spaces

- **Proposition 4.6.8.** *A subgraph H of a graph G is in the cycle space $W_C(G)$ if and only if it has an even number of edges in common with every subgraph in the edge-cut space $W_S(G)$.*
  - ✓ Necessity ($\Rightarrow$) Each subgraph in the cycle space is a union of edge-disjoint cycles, and each subgraph in the edge-cut space is a union of edge-disjoint edge-cuts. Necessity follows from Proposition 4.5.8.
  - ✓ Sufficiency($\Leftarrow$) It may be assumed without loss of generality that G is connected, since the argument that follows may be applied separately to each of the components of G if G is not connected.

Relationship Between the Cycle and Edge-cut Spaces

- ✓ Suppose H has an even number of edges in common with each subgraph in the edge-cut space of G, and let T be a spanning tree of G. Let $e_1, \ldots, e_r$ be the non-tree edges of H, and consider $C = C_1 \oplus \ldots \oplus C_r$, where each C_i is the fundamental cycle in $T + e_i$, $i = 1, \ldots, r$.
- ✓ Arguing as in the proof of Theorem 4.6.5, ***H* ⊕ *C* has no non-tree edges**. Thus, the only possible edges in $H \oplus C$ are edges of T. (If $H \oplus C$ is null, proof completes)
- ✓ So suppose b is an edge in both T and $H \oplus C$. Now if S is the fundamental edge-cut associated with b, then ***b* is the only common edge in *H* ⊕ *C* and *S***.
- ✓ But, since $C \in W_C(G)$, C has an even number of edges in common with each subgraph in the edge-cut space of G, as does H. This implies that $H \oplus C$ has an even number of edges in common with each subgraph in the edge-cut space ($\oplus$: $\text{even}_1 + \text{even}_2 - 2\times\text{edge}_{\text{common}}$).
- ✓ In particular, $H \oplus C$ and S must have an even number of edges in common. this contradiction shows that $H \oplus C$ must be the null graph, that is, $H = C$. Hence, H is a subgraph in the cycle space of G.

Proposition 4.6.9. *A subgraph H of a graph G is an element of the edge-cut space of G if and only if it has an even number of edges in common with every subgraph in the cycle space of G.*

Orthogonality of the Cycle and Edge-Cut Spaces

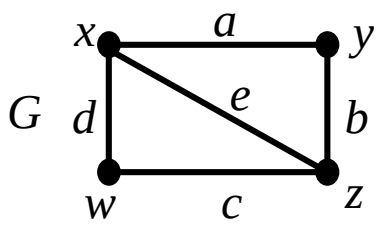
- **DEFINITION**: Two vectors in the edge space $W_E(G)$ are said to be ***orthogonal*** if the dot product of their characteristic vectors equals 0 in $GF(2)$. Thus, two subsets of edges are orthogonal if they have an even number of edges in common.
- **Theorem 4.6.10.** *Given a graph G, the cycle space $W_C(G)$ and edge-cut space $W_S(G)$ are orthogonal subspaces of $W_E(G)$.*
  - ✓ Let H be a subgraph in the cycle space, and let K be a subgraph in the edge-cut space. By either Proposition 4.6.8 or Proposition 4.6.9, H and K have an even number of edges in common and, hence, are orthogonal.
- **Theorem 4.6.11.** *Given a graph G, the cycle space $W_C(G)$ and the edge-cut space $W_S(G)$ are orthogonal complements in $W_E(G)$ if and only if $W_C(G) \cap W_S(G) = \phi$.*
  - ✓ If $W_C(G) \oplus W_S(G)$ denotes the direct sum of the subspaces $W_C(G)$ and $W_S(G)$ then
  - ✓ $dim(W_C(G) \oplus W_S(G)) = dim(W_C(G)) + dim(W_S(G)) - dim(W_C(G) \cap W_S(G))$
  - ✓ By Corollary 4.6.7., $dim(W_C(G)) + dim(W_S(G)) = |E_G|$. Thus, $dim(W_C(G) \oplus W_S(G)) = |E_G| = dim(W_E(G))$ if and only if $dim(W_C(G) \cap W_S(G)) = 0$.

Vector Space of Vertex Subsets

- **FROM LINEAR ALGEBRA:** The column space of a matrix M is the set of column vectors that are linear combinations of the columns of M. When the entries of M come from $GF(2)$, linear combinations are simply mod 2 sums.
- **REVIEW FROM §2.6 :** The incident matrix I_G of G is the matrix whose (i,j)th entry is given by

$$I_G[i,j] = \begin{cases} 0 & \text{if } v_i \text{ is not an endpoint of } e_j \\ 1 & \textit{otherwise} \end{cases}$$

- Analogous to the edge space $W_E(G)$, the collection of vertex subsets of V_G under ring sum forms a vector space over $GF(2)$, which is called the ***vertex space of*** G and is denoted $W_V(G)$.
- Then I_G represents a linear transformation from edge space $W_E(G)$ to vertex space $W_V(G)$, mapping the characteristic vectors of edge subsets to characteristic vectors of vertex subsets.
- **Example 4.6.4:**



$$I_G = \begin{array}{c} \\ x \\ y \\ z \\ w \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

$\{a,c,e\} \to \{y, w\}$ $\qquad$ $\{a,b,c,e\} \to \{z, w\}$

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \qquad \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Path from y to w, $\qquad$ path from z to w

Vector Space of Vertex Subsets

- **NOTATION** : For any edge subset D of a simple graph G, let C_D denote the corresponding set of columns of I_G.
- **Proposition 4.6.12.** *Let D be the edge-set of a cycle in a simple graph G. Then the mod 2 sum of the columns in C_D equals zero.*
  - ✓ For a suitable ordering of the edges and vertices of G, a cycle of length k corresponds to the following submatrix of I_G.

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 1 \\ 1 & 1 & \ddots & \vdots & 0 \\ 0 & 1 & \ddots & 0 & \vdots \\ \vdots & \ddots & \ddots & 1 & 0 \\ 0 & \cdots & 0 & 1 & 1 \end{pmatrix}$$

Vector Space of Vertex Subsets

- **Corollary 4.6.13.** *A set D of edges of a simple graph G forms a cycle or a union of edge-disjoint cycles if and only if the mod 2 sum of the columns in C_D is zero.*
- **Proposition 4.6.14.** *An edge subset D of a simple graph G forms an acyclic subgraph of G if and only if the column set C_D is linearly independent over GF(2).*
- **Proposition 4.6.15.** *Let D be a subset of edges of a connected, simple graph G. Then D forms (includes) a connected spanning subgraph of G if and only if column set C_D spans the column space of I_G.*
  - ✓ The column set C_D spans the column space of I_G if and only if each column of I_G can be expressed as the sum of the columns in a subset of C_D.
  - ✓ But each column of I_G corresponds to some edge $e = xy$ of G, and that column is a sum of the columns in some subset of C_D if and only if there is a path from x to y whose edges are in D.
- **Proposition 4.6.16.** *Let G be a connected, simple graph. Then an edge subset D includes a spanning tree of G if and only if the columns corresponding to the edges of D form a basis for the column space of I_G over GF(2).*
  - ✓ This follows directly form Propositions 4.6.14 and 4.6.15.