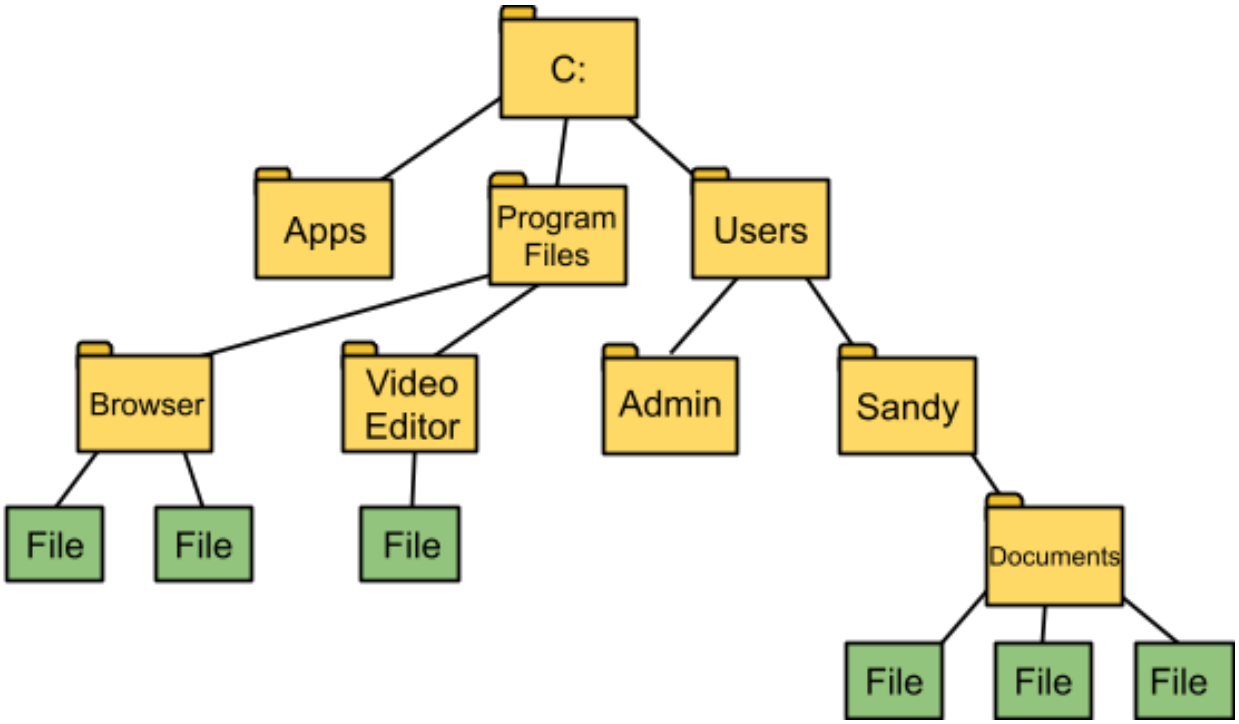
14.1 Introduction to trees

The diagram below shows a portion of a file system on a computer.

Figure 14.1.1: Diagram of a file system.



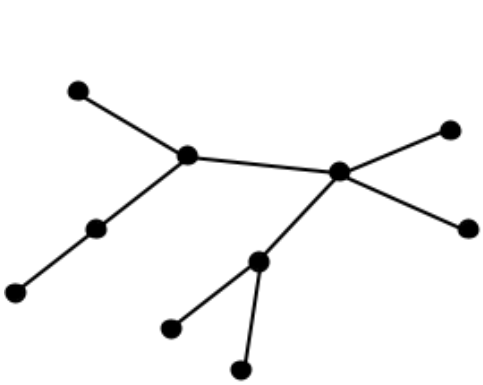
The file system can be seen as a graph in which each folder or file is a vertex. There is an edge between two folders if one folder is a subfolder of the other. There is an edge between a file and a folder if the file resides in that folder. In most computer operating systems, a file or folder can only reside in one location, which means that the underlying graph corresponds to a tree.

Definition 14.1.1: Trees.

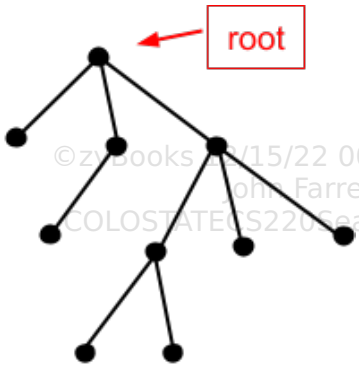
A ***tree*** is an undirected graph that is connected and has no cycles.

The diagram below shows two trees. The two trees are the same, except the vertices in the tree on the right are drawn in a more orderly way.

Figure 14.1.2: A free tree and a rooted tree.



A free tree



A rooted tree

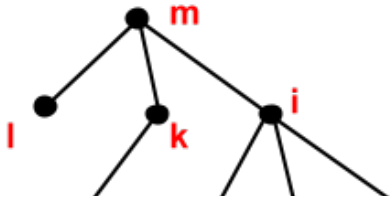
The tree on the left is called a ***free tree*** because there is no particular organization of the vertices and edges. The tree on the right is called a ***rooted tree***. The vertex at the top of the drawing is designated as the ***root*** of the tree. The remaining vertices are organized according to their distance from the root. The distance between two vertices in an undirected graph is the number of edges in the shortest path between the two vertices. The ***level*** of a vertex is its distance from the root. The ***height*** of a tree is the highest level of any vertex. The tree on the right has height 3.

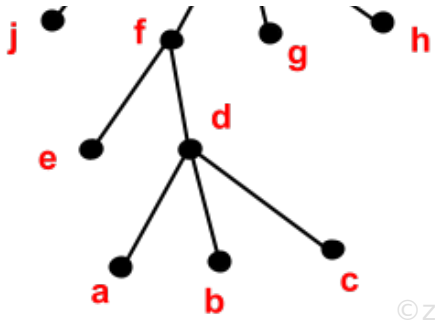
PARTICIPATION ACTIVITY 14.1.1: Free and rooted trees.

Animation captions:

1. In a free tree, the vertices are not ordered. To make a free tree into a rooted tree, select a vertex to be the root and place the root at the top.
2. The root is the only level 0 vertex.
3. The level of each of the remaining vertices is its distance from the root. In the given example, the maximum level is 4, so the tree has height 4.

PARTICIPATION ACTIVITY 14.1.2: Rooted tree example.





1) Which vertex is the root?

Check Show answer

2) What is the height of the tree?

Check Show answer

3) Name a level 3 vertex in the tree.

Check Show answer

The theorem below says that there is exactly one way to travel between every pair of vertices by a path. The proof of the theorem is given elsewhere, along with the other material on the properties of trees.

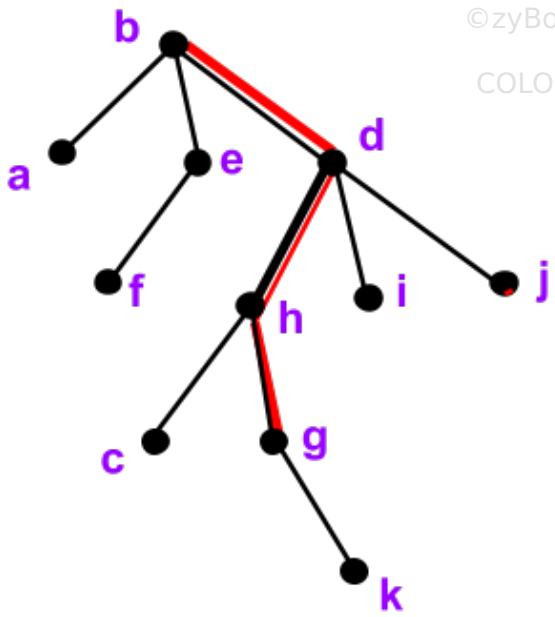
Theorem 14.1.1: Unique paths in trees.

Let T be a tree and let u and v be two vertices in T. There is exactly one path between u and v.

Here are some definitions of terms associated with rooted trees. Each term is defined with respect to the unique path from a vertex to the root.

Definition 14.1.2: Terminology related to rooted trees.

Each definition is illustrated using an example from the rooted tree below. The unique path from g to the root is <g, h, d, b> and is highlighted in red in the figure.



- Every vertex in a rooted tree T has a unique ***parent***, except for the root which does not have a parent. The parent of vertex v is the first vertex after v encountered along the path from v to the root. (Ex: The parent of vertex g is h.)
- Every vertex along the path from v to the root (except for the vertex v itself) is an ***ancestor*** of vertex v. (Ex: The ancestors of vertex g are h, d, and b.)
- If v is the parent of vertex u, then u is a ***child*** of vertex v. (Ex: Vertices c and g are the children of vertex h.)
- If u is an ancestor of v, then v is a ***descendant*** of u. (Ex: The descendants of vertex h are c, g, and k.)
- A ***leaf*** is a vertex which has no children. (Ex: The leaves are a, f, c, k, i, and j.)
- Two vertices are ***siblings*** if they have the same parent. (Ex: Vertices h, i, and j are siblings because they have the same parent, which is vertex d.)
- A ***subtree*** rooted at vertex v is the tree consisting of v and all v's descendants. (Ex: The subtree rooted at h includes h, c, g, and k and the edges between them.)

The animation below reviews the definitions, using a different tree.

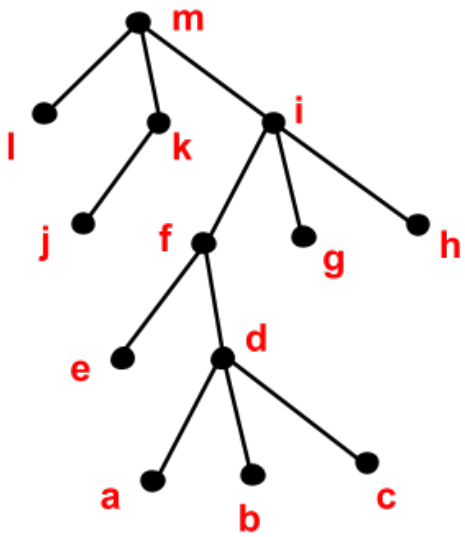
PARTICIPATION ACTIVITY 14.1.3: Terminology for rooted trees.

Animation captions:

1. A rooted tree T is shown. There is an edge {e, i}, and i is below e. e is the parent of vertex i. i is the child of vertex e.
2. The vertex at the top is the root. The root does not have a parent.
3. The leaves are the vertices that do not have any children.
4. Vertices k and l are siblings of each other because they have the same parent.
5. Vertices i, e, and b are the ancestors of vertex m because i, e, and b are the vertices encountered on the path upwards from m.
6. f, g, k, l, and o are on the paths downward from c, so those vertices are the descendants of c. Vertex c and c's descendants form a subtree.

PARTICIPATION ACTIVITY 14.1.4: Terminology for rooted trees.

Here is an example of a rooted tree:



1) Which vertex is the parent of vertex d?

Check Show answer

2) Name a sibling of vertex f.

Check Show answer

3) Name an ancestor of vertex j.

Check Show answer

4) Is vertex e a descendant of vertex h? Type: Yes or No

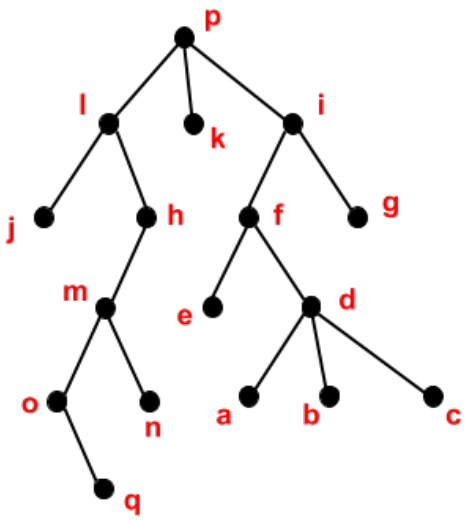
Check Show answer

5) Which of the following vertices is not a leaf: g, h, j, f, c.

Check Show answer

Additional exercises

EXERCISE 14.1.1: Rooted tree basics.



(a) Which vertices are ancestors of vertex n?

(b) Which vertices are the descendants of vertex i?

(c) List the leaves in the tree.

(d) What is the level of vertex d?

(e) What is the height of the tree?

(f) List the level four vertices.

(g) Draw the subtree rooted at vertex h.

(h) What are the siblings of vertex i?

EXERCISE 14.1.2: Enumerating non-isomorphic free trees.

(a) Draw all non-isomorphic free trees with four vertices. Do not label the vertices of the graph. You should not include two trees that are isomorphic.

(b) Draw all non-isomorphic free trees with five vertices. Do not label the vertices of the graph. You should not include two trees that are isomorphic.

EXERCISE | 14.1.3: Mutual descendants in a tree.

(a) Show that if two distinct vertices in a rooted tree, u and v, have a common descendant, then vertex u is a descendant of v or vertex v is a descendant of u. You can assume that every vertex in a rooted tree has a unique path to the root.

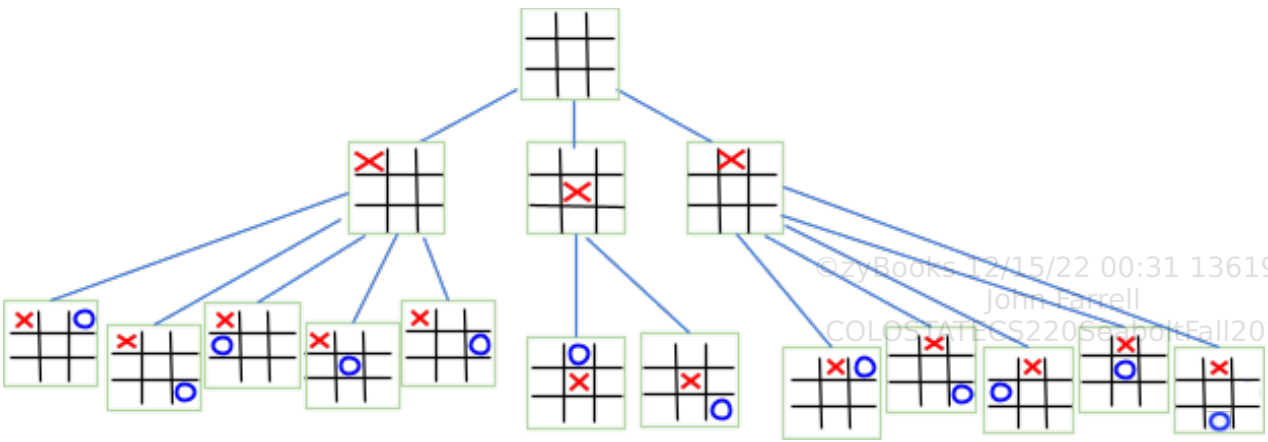
14.2 Tree application examples

Game trees

In 1997, a chess-playing computer program called Deep Blue developed by IBM beat the reigning world champion of chess, Garry Kasparov, in a 6-round series, with two wins, one loss, and three ties. It is not clear whether the victory showed that computers can reason as well as humans, as some claimed. The victory did establish that computers are better at evaluating complex game trees, the basis of computer-based game strategies.

A game tree shows all possible playing strategies of both players in a game. Each vertex in the tree represents a configuration of the game as well as an indication of whose turn it is to play next. For chess, a configuration would be the location of all the pieces still left on the board. We illustrate game trees in a much simpler context: tic-tac-toe. The picture below shows the first two levels of the game tree of tic-tac-toe. (Boards that are equivalent by a rotation or flip are not shown).

Figure 14.2.1: The first three layers of the game tree of tic-tac-toe.



The root of the tree is the initial configuration of the game. In the case of tic-tac-toe, the initial configuration is an empty grid. The odd levels of the game tree represent choices of play for the X

player and the even levels represent choices of play for the O player. The children of a configuration c are all the configurations that can be reached from c by a single move of the correct player. A configuration is a leaf vertex in the tree if the game is over. For tic-tac-toe, the game is over when one of the two players has won or the board is filled up.

The animation below illustrates a portion of the tic-tac-toe game tree near the end of a game.

PARTICIPATION ACTIVITY 14.2.1: A portion of the tic-tac-toe game tree.

Animation captions:

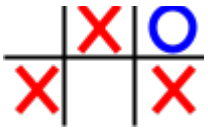
1. The root of the subtree shows a configuration of tic-tac-toe with 3 empty squares. X plays next. One child of the root shows the game with an X placed in an empty square.
2. The two other children of the root show configurations with an X placed in different empty squares. The configurations of the children (level 1 vertices) have 2 empty squares.
3. O plays next. The two children of the first level 1 vertex show an O placed in each of its two empty squares. In one of the children, O has won the game.
4. The children of the second level 1 vertex show an O placed in each of its two empty squares. In both children, O has won the game.
5. The children of the third level 1 vertex show an O placed in each of its two empty squares. In one of the children, O has won the game.
6. The configurations in which O has won are leaves. The other two configurations each have one child which have the board entirely filled and end in a tie.

Theoretically, a game tree can be systematically analyzed to determine optimal playing strategies for each player. However, in practice for most games, the corresponding game tree is much too large to build and analyze in its entirety. Programs like Deep Blue build partial game trees starting from the current configuration in the game and estimate the best next move based on the results of the partial tree.

Tic-tac-toe and chess are examples of deterministic games whose outcome is completely determined by the choices made by each player. A game that involves rolling a pair of dice or shuffling a deck of cards introduces randomization into the game. Probability theory as well as game trees are required to analyze games with an element of chance.

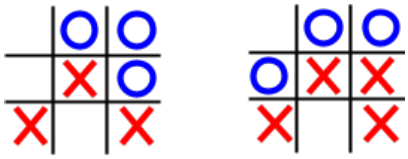
PARTICIPATION ACTIVITY 14.2.2: Tic-tac-toe game tree.

1) Does the shown configuration have a child that is a leaf in the game tree? (X plays next).



Check Show answer

2) Two configurations are shown. Is one a child of the other in the game tree?



Check Show answer

3) A configuration corresponding to a tie game is a leaf in the game tree. What is the level of a tie configuration?

Check Show answer

Prefix codes

Text files are stored on computers as binary strings in which each character is assigned a binary code representing that character. The standard way to represent text is to have a fixed length code for every symbol in the file. ASCII and Unicode are examples of fixed length encodings. More space-efficient encodings can be achieved by ***variable length codes***, in which the number of bits for each character can vary. Characters such as "a" and "e" that occur frequently are represented with fewer bits, while characters such as "z" or "q" are represented using more bits.

Trees are a convenient way to represent variable length codes for translating between text and binary.

PARTICIPATION ACTIVITY 14.2.3: Trees and variable length encoding.

Animation captions:

1. Encode mist. The path from the root to the leaf labeled m encounters five 1's. The encoding begins with 11111.
2. The path from the root to the leaf labeled i encounters the bits 100. The encoding is now 11111100.
3. The path from the root to the leaf labeled s encounters the bits 1110. The encoding is now 111111001110.
4. The path from the root to the leaf labeled t encounters the bits 101. The encoding of mist is 111111001110101.

PARTICIPATION ACTIVITY 14.2.4: Trees and variable length decoding.

Animation captions:

1. Decode 11101010110. The first bit is 1, which says "go right". Follow the edge to the right child of the root.
2. The next bits are 110. Travel right, right, and then left down the tree. A leaf is reached with label s. Output s and go back to the root.
3. The next bits are 101. Travel right, left, and then right down the tree. A leaf is reached with label t. Output t and go back to the root.
4. The next bit is 0. Travel left down the tree. A leaf is reached with label a. Output a and go back to the root.
5. The next bits are 110. Travel right, right, and then left down the tree. A leaf is reached with label r. Output r. The end of the sequence is reached. The output word is star.

Notice that "a" is encoded using just one bit (0), whereas "z" is encoded using 6 bits (111101). Suppose a text message consists of 9,000 a's and 1,000 z's. If each letter were encoded with 3 bits, a total of 10,000*3 = 30,000 bits would be needed. But using the variable length code, only 9,000*1 + 1,000*6 or 15,000 bits would be needed.

The code illustrated in the animation is an example of a prefix code. A string s is a ***prefix*** of another string t if all the characters in string s appear at the beginning of string t. The diagram below illustrates:

Figure 14.2.2: Examples illustrating prefixes.

s: 111
t: 111101

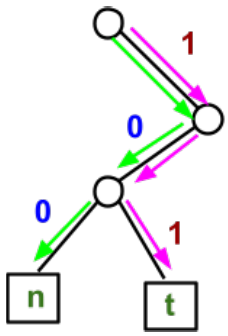
s is a *prefix* of t

r: 1110
t: 111101

r is not a *prefix* of t

A ***prefix code*** has the property that the code for one character *can not* be a prefix of the code for another character. The fact that the codes are organized as a tree in which characters are only stored at the leaves ensures the prefix property. The diagram below illustrates:

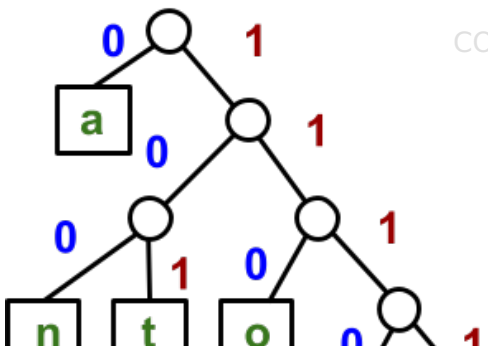
Figure 14.2.3: The prefix property in codes.



Paths to different leaves must eventually diverge.

The prefix property ensures that a binary string can be uniquely decoded with only one pass through the string. By contrast, if the code for "a" is 01 and the code for "b" is 0110, then decoding would be ambiguous. After reading a 01, the decoder would not know whether to output an "a" or read more characters to output a "b".

PARTICIPATION ACTIVITY 14.2.5: Prefix codes.



1) Use the prefix tree above to encode the letter "b".

Check Show answer

2) Use the prefix tree above to encode the string "ba".

Check Show answer

3) Use the prefix tree above to encode the word "bat".

Check Show answer

4) Use the prefix tree above to decode the string: 1110

Check Show answer

5) Use the prefix tree above to decode the string: 1110100

Check Show answer

6) Use the prefix tree above to decode the string: 1110100110101

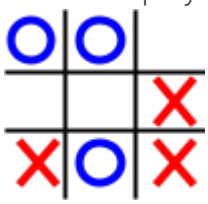
Check Show answer

Additional exercises

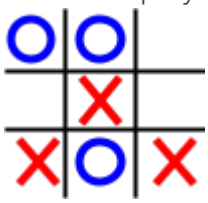
EXERCISE 14.2.1: Configurations of the game tree for tic-tac-toe.

Write down the children of each configuration in the game tree. Indicate which children are leaves and the outcome of the game.

(a) It is O's turn to play next.

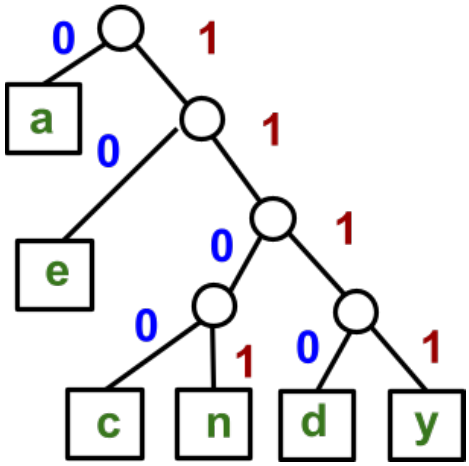


(b) It is X's turn to play next.



EXERCISE | 14.2.2: Encoding and decoding with a prefix code.

Consider the following tree for a prefix code:



(a) Use the tree to encode "day".

(b) Use the tree to encode "candy".

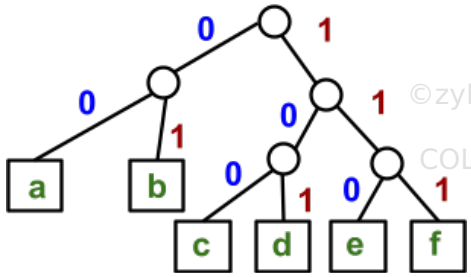
(c) Use the tree to decode "1110101101".

(d) Use the tree to decode "1110011011100 10".

©zyBooks 12/15/22 00:31 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

EXERCISE | 14.2.3: Efficient trees for prefix codes.

Consider the following tree for a prefix code:



A text file contains only characters from the set {a, b, c, d, e, f}. The frequency of each letter in the file is:

- a: 5%
- b: 5%
- c: 10%
- d: 15%
- e: 25%
- f: 40%

(a) What is the average number of bits per character used in encoding the file?

(b) Is there a prefix tree for the set {a, b, c, d, e, f} that would result in fewer bits per character on average for the given frequencies of the characters?

14.3 Properties of trees

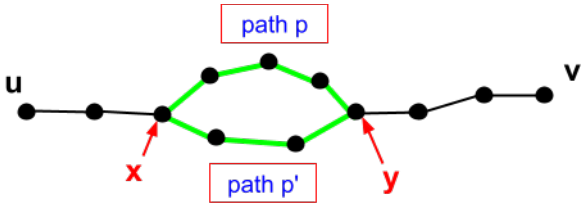
The definitions for parent and child vertices in rooted trees make use of the fact that every vertex has a unique path to the root. There is, in fact, a unique path between any pair of vertices in a tree. The theorem below applies to free trees as well as rooted trees.

Theorem 14.3.1: Unique paths in trees.

Theorem: There is a unique path between every pair of vertices in a tree.

Proof 14.3.1: Unique paths in trees.

Proof. A tree is defined to be a connected graph with no cycles. Because every tree is connected, there is at least one path between every pair of vertices. It remains to establish that there is at most one path between every pair of vertices. The proof is by contrapositive. We assume that there is a pair of vertices u and v such that there are two distinct paths between u and v and then show that the graph must have a cycle (and therefore can not be a tree).



Let p and p' be the two distinct paths between u and v. Find the first place where the two paths differ. Let x be the vertex just before the point where the two paths diverge. Follow path p until it hits a vertex y that is also contained in p'. (Both paths end up at v, so there has to be a point where the two paths come together). The portion of the path p between x and y and the portion of the path p' between x and y form a cycle. ■

PARTICIPATION ACTIVITY 14.3.1: Distinct paths and cycles.

1) The two sequences of nodes given below denote two distinct paths in a graph between vertices j and d.

$$\langle j, h, b, i, g, f, c, d \rangle$$
$$\langle j, h, b, a, f, e, c, d \rangle$$

Which of the following answers describes a cycle that you can deduce is in G?

- ○ $\langle b, a, g, i, b \rangle$
- ○ $\langle b, a, f, g, i, b \rangle$
- ○ $\langle b, a, f, e, c, f, g, i, b \rangle$

The term leaf is defined in rooted trees as a vertex with no children. A leaf has a natural definition in free trees as well. A ***leaf*** of a free tree is a vertex of degree 1. There is one technicality: if a free tree

has only one vertex, then that vertex is a leaf. A vertex is an ***internal vertex*** if the vertex has degree at least two. The following theorem gives a lower bound on the number of leaves in a free tree:

Theorem 14.3.2: Number of leaves in a tree.

Theorem: Any free tree with at least two vertices has at least two leaves.

Proof 14.3.2: Number of leaves in a tree.

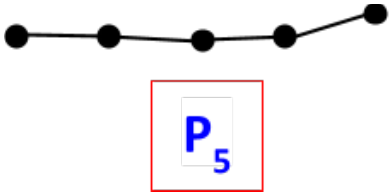
Proof.
Find the longest path p in the tree. Let u and v be the first and last vertices in the path. Suppose u is not a degree 1 vertex. (The same argument will hold for v). Then there is an edge e that is incident to u and not part of the path p.

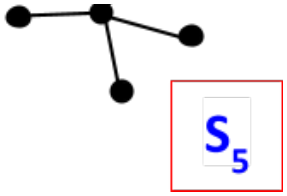
If the other endpoint of e, that is not vertex u, is part of the path p, then the graph has a cycle and is not a tree. The path p can be extended by adding e along with the other endpoint of e. The fact that the path p can be extended contradicts the assumption that p was the longest path in the tree.

We have shown that the two endpoints in the path p are both leaves. Therefore, the tree has at least two leaves. ■

PARTICIPATION ACTIVITY 14.3.2: Stars and paths.

This question relates to two classes of graphs. The first class is the set of graphs which consist of a single path. Let P_n be the graph that consists only of a path with n vertices. A picture of P_5 is given below on the left. The second class of graphs are called star graphs. S_k consists of a single vertex connected by an edge to k other vertices. A picture of S_5 is given below on the right.





1) Is P_n a tree? Type: Yes or No

Check Show answer

2) If $n \geq 2$, then how many leaves are there in P_n?

Check Show answer

3) How many edges are there in P_n? Express your answer as a function of n.

Check Show answer

4) Is S_k a tree? Type: Yes or No

Check Show answer

5) How many vertices are there in S_k? Express your answer as a function of k.

Check Show answer

6) How many edges are there in a star graph with n vertices? Express your answer as a

function of n

Check Show answer

The previous question explores two very different classes of trees (paths and stars) and establishes that each path or star with n vertices has n - 1 edges. The theorem below generalizes the fact to any tree.

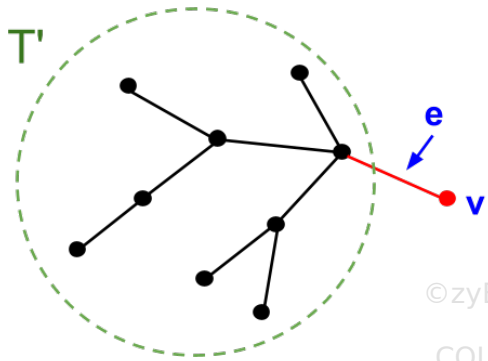
Theorem 14.3.3: Number of edges in a tree.

Theorem: Let T be a tree with n vertices and m edges, then m = n - 1.

Proof 14.3.3: Number of edges in a tree.

Proof. The proof is by induction on the number of vertices. The base case is where n = 1. If T has one vertex, then it is has no edges. Then m = 0 = n - 1.

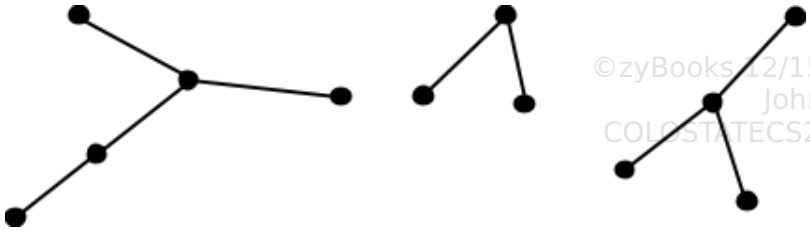
For the inductive step, assume the theorem holds for trees with n-1 vertices and prove that it holds for trees with n vertices. Consider an arbitrary tree T with n vertices. Since n ≥ 2, by the previous theorem, the tree has at least two leaves. Let v be one of the leaves. Remove v from T along with the edge e incident to v. The resulting graph (call it T') is also a tree and has n-1 vertices.



By the induction hypothesis, The number of edges in T' is (n - 1) - 1 = n - 2. T has exactly one more edge than T', because only edge e was removed from T to get T'. Therefore the number of edges in T is n - 2 + 1 = n - 1.■

PARTICIPATION ACTIVITY 14.3.3: Number of edges in a forest.

A ***forest*** is a graph that has no cycles and that is not necessarily connected. The picture below shows a forest with 3 connected components.



1) How many vertices are there in the forest pictured above?

Check Show answer

2) How many edges are there in the forest pictured above?

Check Show answer

3) Suppose that there is a forest with n vertices and c connected components. Give a conjecture for the number of edges in the forest as a function of n and c. Type an expression like: n + c

Check Show answer

Additional exercises

EXERCISE 14.3.1: Finding a graph with a given set of properties.

Find a graph with the given set of properties or explain why no such graph can exist.

(a) Tree, seven vertices, total degree = 14.

(b) Connected, six vertices, six edges.

(c) Tree, four internal vertices, four leaves.

(d) Acyclic, seven vertices, five edges.

(e) Tree, all vertices have degree 2.

EXERCISE 14.3.2: Trees with the most leaves.

(a) If T is a tree with n vertices, what is the most leaves that it can have? Your answer will be an expression involving the variable n. Explain your reasoning. Be sure to address small values for n (e.g., n = 1 or 2).

(b) Draw a tree with eight vertices that has the most number of leaves possible.

EXERCISE 14.3.3: Graphs with n - 1 edges.

(a) We know that if a graph is a tree with n vertices and m edges, then m = n - 1. Is the converse true? That is if a graph has n vertices and n - 1 edges can you conclude that it is a tree? Justify your answer.

EXERCISE 14.3.4: Number of edges in a forest.

(a) Prove that a forest with c components and n vertices has n - c edges. Assume $n \geq c$. (Hint: the base case will have n = c.)

(b) Use the fact proven in the previous problem to show that if a graph has no cycles and has at least n - 1 edges then the graph must be connected.

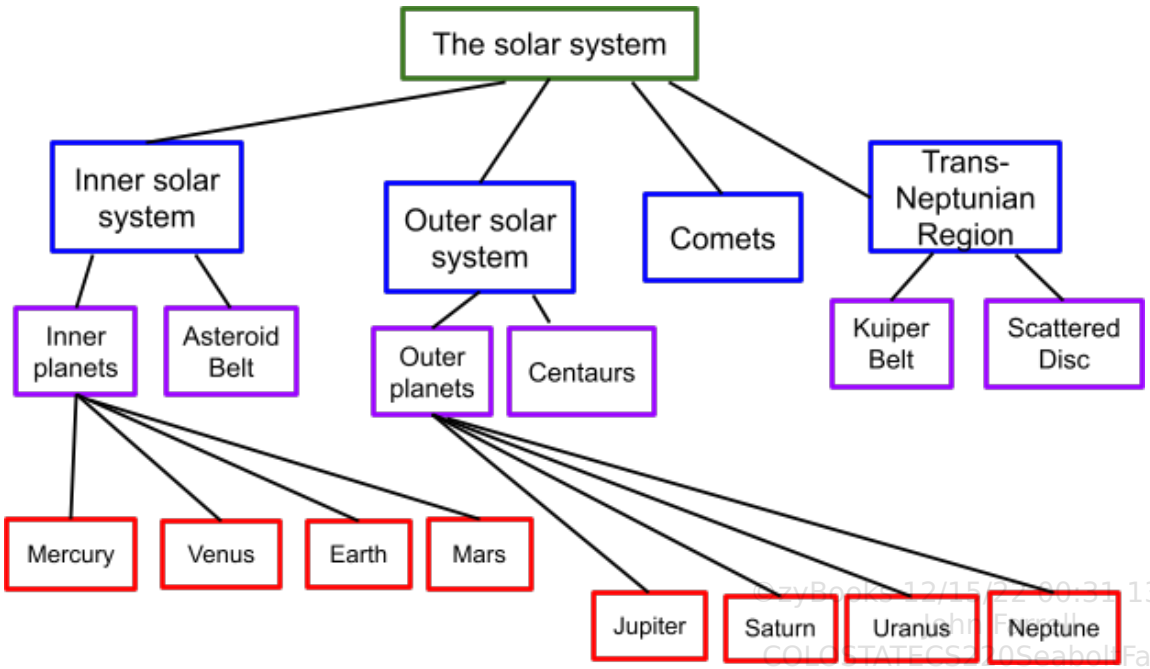
EXERCISE | 14.3.5: Number of edges and connectivity.

(a) Let G be an undirected graph with n vertices and m edges. Prove that if m < n - 1, then G is not connected.

14.4 Tree traversals

Trees are used in computer programs to organize and store information. A common procedure performed on a tree (called a tree ***traversal***) is to process the information stored in the vertices by systematically visiting each vertex. As each vertex is visited, a processing step is performed which might entail a computation or outputting some data. For example, a tree could be used to hold the hierarchical structure of the sections of a book. In order to print out the table of contents, as each vertex is visited, the title of the corresponding portion of the book is printed out. The diagram is a tree that represents the outline of a book or talk on the solar system:

Figure 14.4.1: Tree storing an outline.



The order in which the vertices are visited depends on the application. We present two common tree traversal algorithms: pre-order traversal and post-order traversal. In a ***pre-order traversal***, a vertex is visited before its descendants. In a post-order traversal, a vertex is visited after its descendants. In both traversal algorithms, once a vertex of a subtree is visited, all of the vertices in

that subtree are visited before vertices outside the subtree. The pseudocode for both traversal algorithms is given below. Each procedure would be triggered by an initial call with the root as the input parameter. The step "process(v)" is a generic step that would be replaced by a particular operation that depends on the application.

Figure 14.4.2: Pseudocode for pre-order traversal.


```
Pre-order(v)

process(v)
For every child w of
v:
    Pre-order(w)
End-for
```

Figure 14.4.3: Pseudocode for post-order traversal.

```
Post-order(v)

For every child w of
v:
    Post-order(w)
End-for
process(v)
```

Both the pre-order and post-order traversal algorithms depend on an ordering imposed on the children of each vertex. In the solar system example, the children of a vertex are ordered according to their distance from the sun. The order of the children dictates the order in which each subtree rooted at each child is visited. In this material, the ordering of the children is handled somewhat informally by ordering the children of a vertex from left to right as drawn on the page/screen. A tree represented in a computer would likely store an ordered list of children for each vertex. The particular ordering would depend on how the tree is used.

The animation below illustrates a pre-order traversal using a subset of the solar system tree. The process(v) step in the generic pre-order algorithm shown above is replaced by a statement that prints out the name of the topic stored at vertex v. There are some additional "indent" and

"unindent" commands for formatting the outline nicely. The result is a printed outline of the information about the solar system.

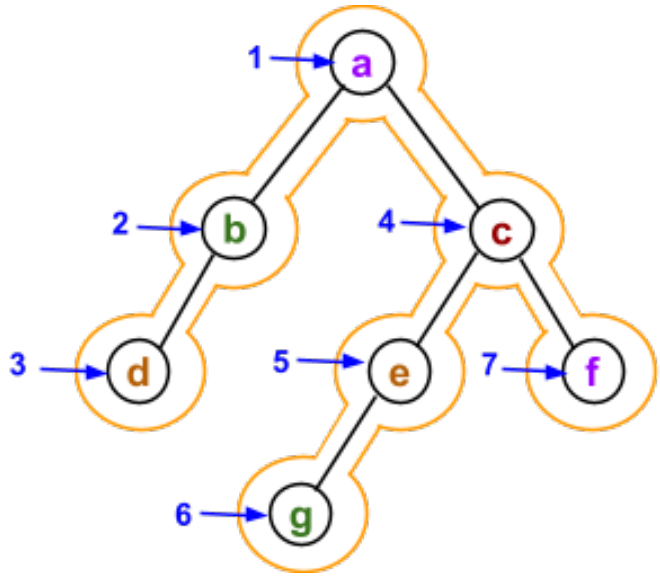
PARTICIPATION ACTIVITY 14.4.1: Pre-order tree traversal.

Animation captions:

1. Preorder(v) is called with v = "The Solar System". v's name is printed.
2. The first child of the current v is "Inner Solar System". Preorder(v) is called with v = "Inner Solar System".
3. v's name is printed. The first child of the current v is "Inner Planets". Preorder(v) is called with v = "Inner Planets".
4. v's name is printed. "Inner Planets" has no children in the tree, so Preorder returns.
5. v is now "Inner Solar System" again. Preorder(v) is called with v equal to the next child of "Inner Solar System" which is "Asteroid Belt".
6. "Asteroid Belt" is printed and Preorder returns because "Asteroid Belt" has no children in the tree.
7. The current v is "Inner Solar System" again. All the leaves of the current v have been visited, so Preorder returns.
8. The current v is the root, "The Solar System", again. The right subtree of the root is traversed and printed. The initial call to the root returns and the pre-order traversal is finished.

Here is a useful trick for determining the order in which the vertices of a tree are visited in a pre-order traversal. Draw an outline all the way around the tree (shown below in orange). Starting at the root and moving to the left, follow the outline all the way around the tree. When you pass the left side of a vertex, visit that vertex.

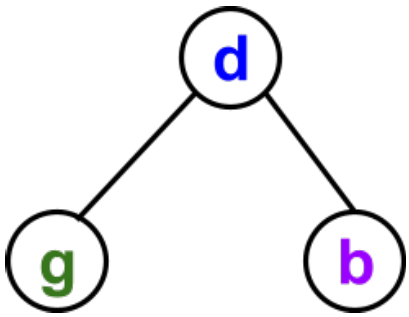
Figure 14.4.4: How to find the pre-order traversal of a tree.



Pre-order traversal:
a b d c e g f

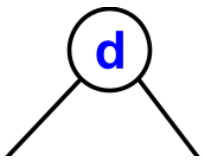
PARTICIPATION ACTIVITY 14.4.2: Pre-order traversal on small trees.

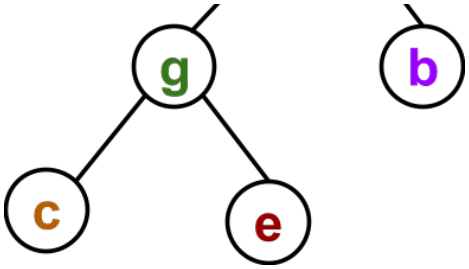
1) In what order are the vertices visited in a pre-order traversal of the tree below?



- g, d, b
- d, g, b
- g, b, d

2) In what order are the vertices visited in a pre-order traversal of the tree below?

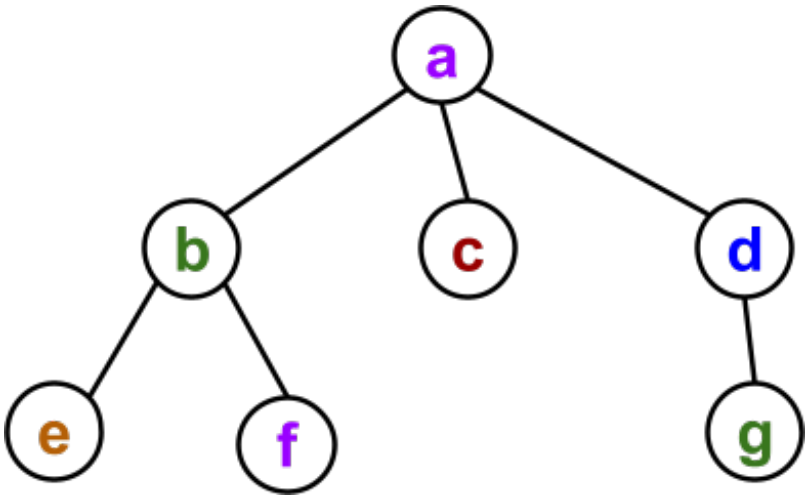




- d, g, b, c, e
- d, c, g, e, b
- d, g, c, e, b

PARTICIPATION ACTIVITY 14.4.3: Pre-order traversal of a tree.

Give the ordering that the vertices are visited according to a pre-order traversal of the tree below.



If unable to drag and drop, refresh the page.

b d a g c f e

1

2

3

4

5

6

7

Reset

Post-order traversal

The animation below shows a procedure to count the number of leaves in a tree using a post-order traversal. The process(v) step is replaced by a few lines that assign a value called leaf-count to each vertex. The procedure assigns the number of leaves in the subtree rooted at v to leaf-count(v). If the vertex is a leaf, then leaf-count = 1. If the vertex is not a leaf, leaf-count is assigned a value equal to the sum of the leaf-counts of the children. The final value assigned to the root's leaf-count is the total number of leaves in the tree.

A post-order traversal is appropriate for counting the number of leaves in a tree because the leaves must be counted for each subtree rooted at the children of v before they can be counted for the subtree rooted at v. Post-order traversals are commonly used for operations that compute a function of the data stored over an entire tree or subtree, such as evaluating player strategies in game trees.

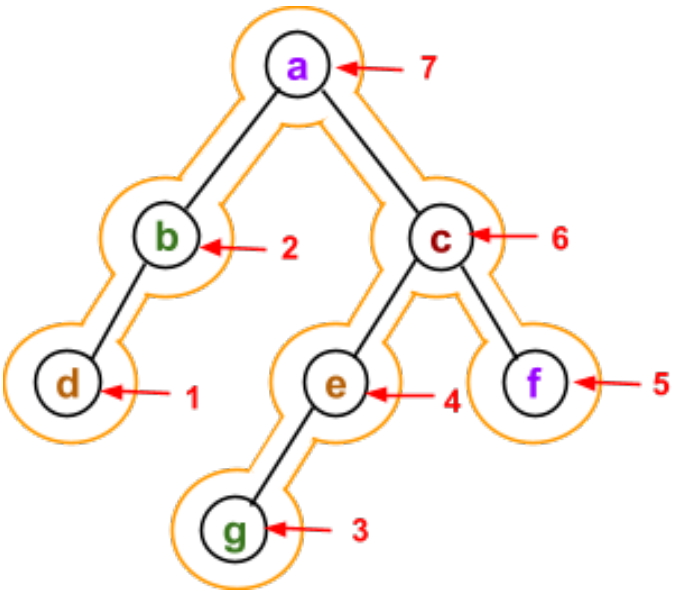
PARTICIPATION ACTIVITY 14.4.4: Post-order tree traversal.

Animation captions:

1. post-order(v) is called with v = a, the root of the tree. The first child of a is b, so post-order(b) is called. b has no children so the For-loop is skipped.
2. Since b is a leaf, the leaf-count(b) is set to 1. post-order returns. v is now a again. The next child of a is c, so there is a call to post-order(c).
3. The first child of c is d. post-order(d) is called.
4. Since d is a leaf, the For-loop is skipped and leaf-count(d) is set to 1. post-order returns.
5. The current value of v is c again. The next child of c is e. post-order(e) is called.
6. Since e is a leaf, the For-loop is skipped and leaf-count(e) is set to 1. post-order returns.
7. The current value of v is c again. The next child of c is f. post-order(f) is called. Since f is a leaf, leaf-count(f) is set to 1. post-order returns.
8. v is c again. All of c's children are done. Since c is not a leaf, leaf-count(c) is the sum of the leaf-count value of c's children: 1 + 1 + 1 = 3. post-order returns.
9. v is a again. All of a's children are done. leaf-count(a) is the sum of the values of a's children: 1 + 3 = 4. post-order returns. post-order is done.

The order in which the vertices are visited in a post-order traversal can be determined using a similar trick as with a pre-order traversal. Draw an outline all the way around the tree (shown below in orange). Starting at the root and moving to the left, follow the outline all the way around the tree. When you pass the right side of a vertex, visit that vertex.

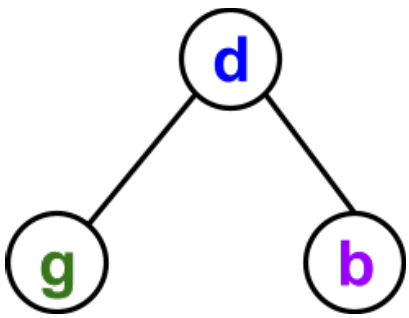
Figure 14.4.5: How to find the post-order traversal of a tree.



Post-order traversal:
d b g e f c a

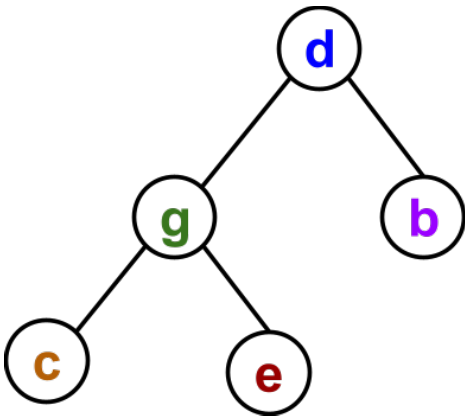
PARTICIPATION ACTIVITY 14.4.5: Post-order traversal on small trees.

1) In what order are the vertices visited in a post-order traversal of the tree below?



- g, d, b
- d, g, b
- g, b, d

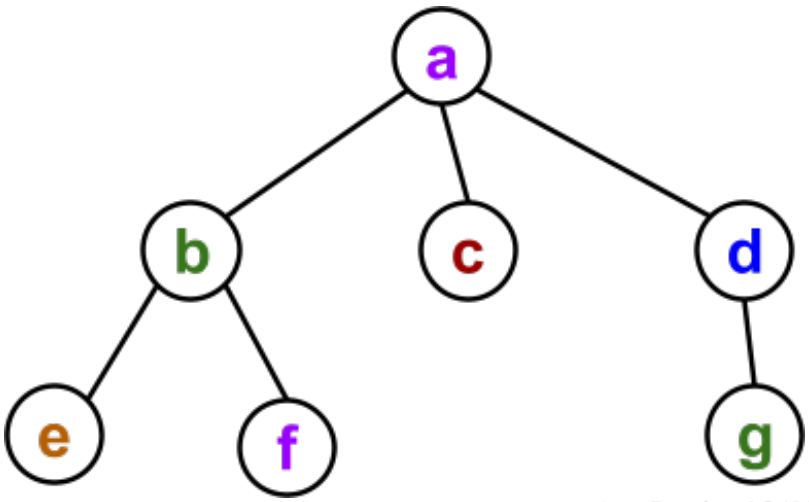
2) In what order are the vertices visited in a post-order traversal of the tree below?



- c, e, b, g, d
- g, c, e, b, d
- c, e, g, b, d

PARTICIPATION ACTIVITY 14.4.6: Post-order traversal of a tree.

Give the ordering that the vertices are visited according to a post-order traversal of the tree below.



If unable to drag and drop, refresh the page.

b e c f g a d

2

3

4

5

6

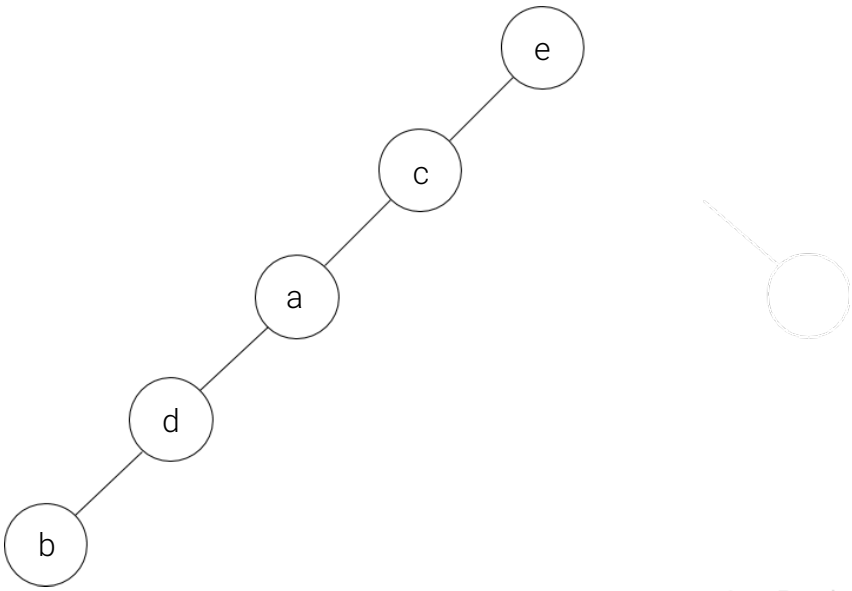
7

Reset

CHALLENGE ACTIVITY 14.4.1: Tree traversals.

422102.2723990.qx3zqy7

Start



Enter the labels of each vertex in the order visited in a pre-order traversal.

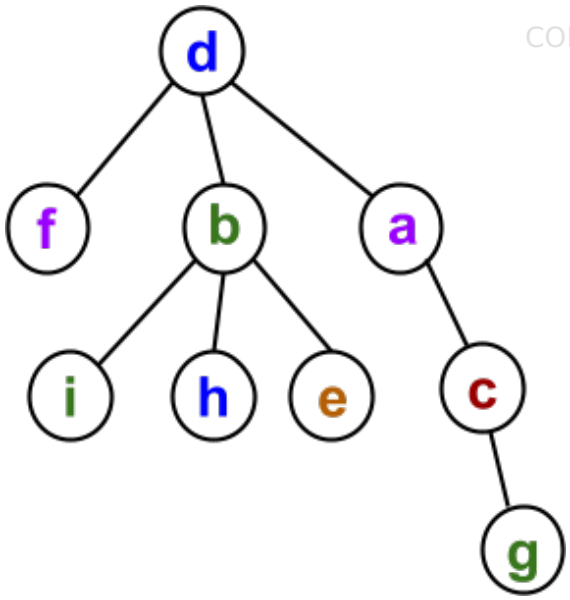
Ex: a, b, c

1	2	3	4

Check Next

Additional exercises

EXERCISE | 14.4.1: Ordering vertices according to post-order and pre-order traversals.



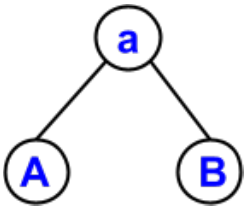
(a) Give the order in which the vertices of the tree are visited in a post-order traversal.

(b) Give the order in which the vertices of the tree are visited in a pre-order traversal.

©zyBooks 12/15/22 00:31 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

EXERCISE 14.4.2: Trees that correspond to pre-order traversals.

Each sequence represents a pre-order traversal of a tree. Each letter is the label of a vertex. Internal vertices are labeled with lower-case letters and leaves are labeled with capital letters. Furthermore, the tree has the property that every internal vertex has exactly two children. Give a tree T such that a pre-order traversal of T visits the vertices in the order given. For example, the sequence a, A, B would correspond to the tree:



(a) a, b, A, B, C

(b) a, b, c, A, B, C, d, D, e, E, F

(c) If the number of children of an internal vertex can be one or two, does each sequence correspond to a unique tree? That is, is it possible to have two different trees whose pre-order traversals result in the same sequence? (Capital letters must be leaves and lower-case letters must be internal vertices.)

EXERCISE 14.4.3: Adapting post-order traversal to compute properties of a tree.

Below is the pseudo-code for a post-order traversal of a tree.

```
Post-order(v)

For every child w of v:
    Post-order(w)
End-for
process(v)
```

(a) Adapt the code to compute the total number of vertices in the tree. The algorithm can return a value.

(b) Adapt the code to compute the height of the tree. The algorithm can return a value.

14.5 Spanning trees and graph traversals

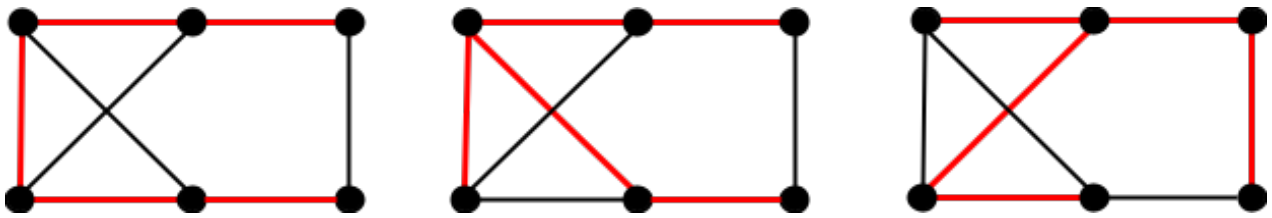
Suppose an application needs to broadcast information to every computer (vertex) in a network. The information will be disseminated along the communication links so that every vertex in the network receives the information. Moreover, the goal is to minimize overall congestion, so it would be wasteful to send information along alternate paths to the same location. The information should be broadcast over a spanning tree of the network. Here is the formal definition of a spanning tree:

Definition 14.5.1: Spanning tree of a connected graph.

A ***spanning tree*** of a connected graph G is a subgraph of G which contains all the vertices in G and is a tree.

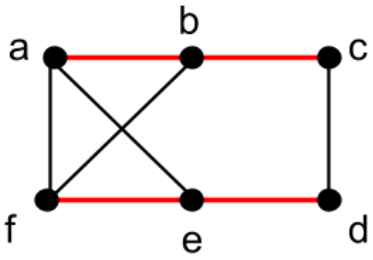
The picture below shows several possible spanning trees of the same graph. The edges of the spanning tree are shown in red. The vertex set of the spanning tree is always the entire set of vertices in the graph.

Figure 14.5.1: Three different spanning trees of a graph.



PARTICIPATION ACTIVITY 14.5.1: Completing a spanning tree with an edge.

The graph below shows a partially constructed spanning tree. The edges in red are included in the partial tree.



1) Which edge, when added, will create a spanning tree?

- {a,e} only
- {a,f} only
- {c,d} only
- Any of the above

There are two common methods for finding spanning trees in a graph: ***Breadth-First Search*** and ***Depth-First Search***. Both methods start at a single vertex and incrementally build a connected tree by adding edges and vertices. The resulting tree is rooted at the start vertex. ***Graph traversal*** is a process that systematically explores all the vertices of a graph. Breadth-first search and depth-first search are used as a subroutine for traversing a graph in many other graph algorithms.

As the name suggests, Depth-First Search (DFS) favors going deep into the graph and tends to produce trees with longer paths from the start vertex. Breadth-First Search (BFS) explores the graph by distance from the initial vertex, starting with its neighbors and expanding the tree to neighbors of neighbors, etc. If you think of each algorithm as an explorer who is visiting each vertex in a graph, the DFS explorer sets out along new paths, always looking to go far beyond her starting location. The BFS explorer, on the other hand, is more conservative, favoring locations close to her initial location before venturing beyond. A breadth-first search tree is likely the preferred spanning tree for broadcasting in a network because it produces the shortest paths from the initial (source) vertex.

Depth-First Search

The execution of depth-first search is illustrated in the animation below. Although the edges of the spanning tree T and the underlying graph G are undirected, the animation shows the direction that an edge is traversed as it is added to T. The explorer backtracks when there are no unexplored neighbors at her current location. She always backtracks along the edge which was used to arrive at her original location. That is, she backtracks in the reverse direction of an edge:

PARTICIPATION ACTIVITY 14.5.2: Depth-first search.

Animation captions:

1. Start at vertex a. Vertex a's neighbors are b and g.
2. Go to b first. Edge {a, b} is added to the DFS tree T.
3. b's neighbors are a, c, and g. Vertex a has already been visited, so go to c. Add {b, c} to T.
4. Go to c's next unvisited neighbor, which is d. Add {c, d} to T. d's only neighbor is c, so d has no more unvisited neighbors. Backtrack to c.
5. c's neighbors are b, d, e, and h. Vertex b and d have already been visited, so go to e. Add {c, e} to T.
6. e's neighbors are c and f. Vertex c has already been visited, so go to f. Add {e, f} to T.

7. f's neighbors are e, g, and h. Vertex e has already been visited, so go to g. Add {f, g} to T.
8. All g's neighbors have been visited. Backtrack to f. Visit h from vertex f. Add {f, h} to T. All vertices have been visited, so keep backtracking to start vertex a.

The pseudocode for depth-first search is given below. There is an outer procedure that calls a recursive procedure named visit. The recursive calls keep track of where to backtrack when all the neighbors of the current vertex have already been added to T. The animation that follows repeats the execution of depth-first search along with the pseudocode for the procedure.

Figure 14.5.2: Depth-first search.

```
Depth-first-search

Input: An undirected, connected graph G. A start
vertex v1
Output: T, a depth-first search tree.

Add v1 to T.
visit(v1)

visit(v)

For every neighbor w of v:
    If w is not already in T
        Add w and {w, v} to T.
        visit(w);
    End-if
End-for
```

Note that there is some ambiguity with regard to the order in which the neighbors of a vertex are considered. For the examples presented here, neighbors are considered in alphabetical order.

PARTICIPATION ACTIVITY 14.5.3: Depth-first search with pseudocode.

Animation content:

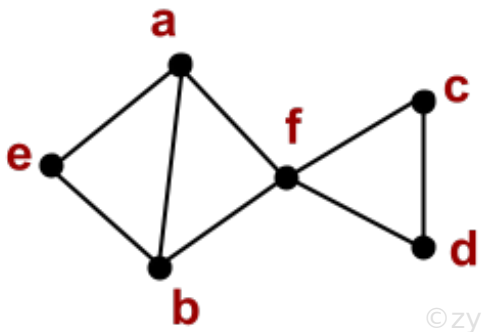
undefined

Animation captions:

1. Call visit(a). Add visit(a) to stack. Vertex b is the first neighbor of a. Since b is not in T, add vertex b and edge {a,b} to T. Call visit(b). Add visit(b) to the stack.
2. The first neighbor of b that is not already in T is c. Add vertex c and edge {b, c} to T. Call visit(c). Add visit(c) to the stack.
3. The first neighbor of c that is not already in T is d. Add vertex d and edge {c, d} to T. Call visit(d). Add visit(d) to the stack.
4. All d's neighbors are in T, so return from visit(d). Pop visit(d) from the stack. The current call at the top of the stack is visit(c).
5. The next neighbor of c that is not already in T is e. Add vertex e and edge {c, e} to T. Call visit(e). Add visit(e) to the stack.
6. The next neighbor of e that is not already in T is f. Add vertex f and edge {e, f} to T. Call visit(f). Add visit(f) to the stack.
7. The next neighbor of f that is not already in T is g. Add vertex g and edge {f, g} to T. Call visit(g). Add visit(g) to the stack.
8. All g's neighbors are in T. Return from visit(g) to visit(f). The next neighbor of f that is not in T is h. Add vertex h and edge {f, h} to T. Call visit(h).
9. All h's neighbors are in T. Return from visit(h) to visit(f). All vertices are in T, so each call returns to the previous call until visit(a) returns and DFS is complete.

PARTICIPATION ACTIVITY 14.5.4: Determining a DFS tree for a graph.

Consider an execution of depth-first search that starts at vertex a. Select the edges that are included in the depth-first search tree in the order that they are added to the tree.



If unable to drag and drop, refresh the page.

{b,e} **{f, c}** **{b,f}** **{c,d}** **{a,b}**

1

2

3

4

5

Reset

Breadth-first-search

Breadth-first-search visits vertices in the graph according to their proximity to the start vertex. The algorithm maintains a list of vertices to be visited soon. Vertices are added to the back of the list and the next vertex to visit is taken from the front of the list. The animation below illustrates the execution of breadth-first search on a sample graph:

PARTICIPATION ACTIVITY 14.5.5: Breadth-first search.

Animation captions:

1. Start at vertex a. The current vertex is a. The list is empty.
2. Add neighbors of a (b and g) to the list. {a, b} and {a, g} are added to the BFS tree T.
3. Take the first vertex b off the list and make b the current vertex. Add neighbors of b that have not yet been added (vertex c) to the list. Add {b, c} to T.
4. Remove the next vertex g and make g the current vertex. Add neighbors of g that have not yet been added (vertex f) to the list. Add {g, f} to T.
5. Remove c and make c the current vertex. Add neighbors of c that have not yet been added (vertices d, e, and h) to the list. Add {c, d}, {c, e}, and {c, h} to T.
6. Remove the next vertex f and make f the current vertex. Vertex f has no neighbors that have not been added.
7. Since none of the vertices on the list have unadded neighbors, each is removed and no new vertices are added. When the list is empty, BFS is done.

Here is the pseudo-code for breadth-first search:

Figure 14.5.3: Breadth-first search.

Breadth-first-search

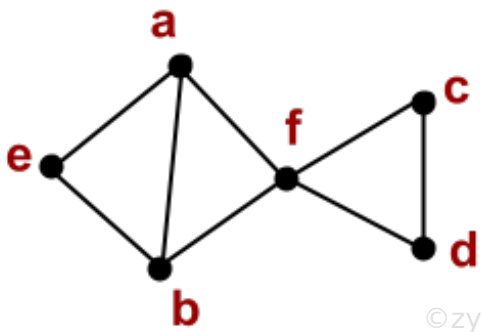
Input: An undirected, connected graph G. A start vertex v_1
Output: T, a breadth-first search tree.

Add v_1 to T.
Add v_1 to the back of the list.

While the list is not empty:
    Remove vertex v from the front of the list.
    For each neighbor w of v that is not already in T:
        Add w and {w, v} to T.
        Insert w at the back of the list.
    End-for
End-while

PARTICIPATION ACTIVITY 14.5.6: Determining the structure of a BFS tree.

Consider breadth-first search run on the graph below starting from vertex a.



In the resulting BFS tree rooted at vertex a, vertex v is a child of vertex u if edge {u, v} and vertex v are added when vertex u is removed from the list.

1) How many children does vertex a have in the resulting BFS tree?

Check Show answer

2) How many children does vertex b have in the resulting BFS tree?

Check Show answer

3) How many children does vertex e have in the resulting BFS tree?

Check Show answer

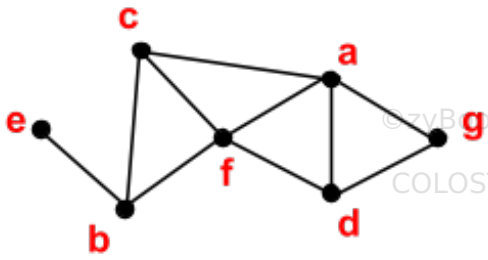
4) How many children does vertex f have in the resulting BFS tree?

Check Show answer

Additional exercises

EXERCISE 14.5.1: BFS and DFS trees for a small graph.

(a) Give the tree resulting from a traversal of the graph below starting at vertex a using BFS and DFS. Assume that the neighbors of a vertex are considered in alphabetical order.

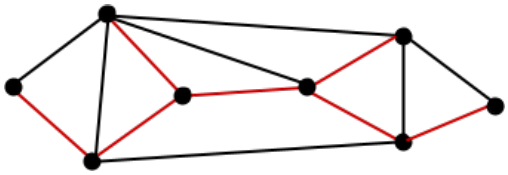


EXERCISE 14.5.2: When are a BFS tree and a DFS tree the same?

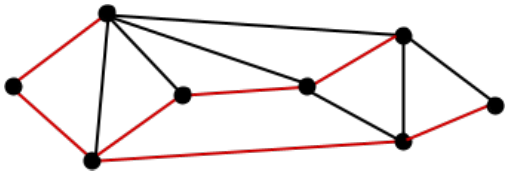
(a) Give an example of a graph with five vertices along with a labeling of the vertices (a, b, c, d, e) so that the resulting BFS tree and DFS tree are the same and the edges are added to the tree in the same order. Assume that for both BFS and DFS, the neighbors of a vertex are considered in alphabetical order and both start at vertex a.

EXERCISE 14.5.3: Labeling vertices to produce a particular BFS or DFS tree.

(a) Label the vertices of the graph given below so that if BFS is run starting at vertex a and the neighbors are considered in alphabetical order, then the resulting BFS tree consists of the edges shown in red.



(b) Label the vertices of the graph given below so that if DFS is run starting at vertex a and the neighbors are considered in alphabetical order, then the resulting DFS tree consists of the edges shown in red.



EXERCISE 14.5.4: When is an edge in every spanning tree?

(a) Under what circumstances does an edge in a connected graph belong to every spanning tree?

EXERCISE 14.5.5: Edges between descendants in a BFS tree.

(a) Consider a tree T resulting from running breadth-first search on an undirected graph. Is it possible to have an edge between a vertex v and a descendant of v that is not included in T? Why or why not?

EXERCISE | 14.5.6: Adapting DFS for cycle detection.

(a) Depth-first search can be used to detect whether a graph has any cycles. How would you augment the pseudo-code for depth-first search, given below, to determine if the graph has a cycle? You can assume that the graph is connected.

```
visit(v)

For every neighbor w of v:
    If w is not already in T
        Add w and {w, v} to T.
        visit(w);
    End-if
End-for
```


14.6 Minimum spanning trees

A spanning tree of a graph specifies a subset of the edges that provides connectivity between every pair of vertices while minimizing the number of edges included. Every spanning tree of a graph with n nodes has n-1 edges, so there is no reason to prefer one spanning tree over another if the only goal is to include as few edges as possible.

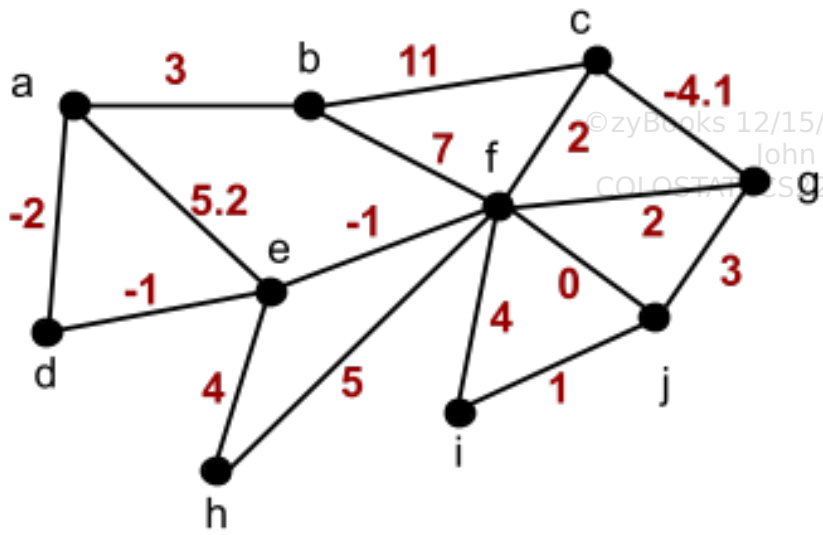
Suppose, alternatively, that different edges have different costs. For example, in broadcasting a message across a network, the sender may be required to pay for bandwidth. Different communication links could have different costs depending on their length, congestion conditions or the quality of the channel. When edges have varying costs, a natural goal is to minimize the total cost of the spanning tree. Some definitions are required to formalize the problem.

Definition 14.6.1: Weighted graph.

A ***weighted graph*** is a graph G = (V ,E), along with a function w: E → **R**. The function w assigns a real number to every edge.

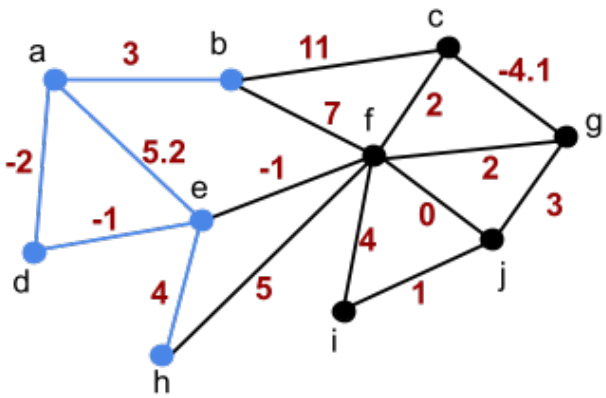
A weighted graph is pictured below. The weight of each edge is drawn next to the edge.

Figure 14.6.1: A weighted graph.



If H is a subgraph of a weighted graph G, the weight of H (denoted w(H)) is the sum of the weights of the edges in H. The diagram below shows a subgraph of a weighted graph and the weight of the subgraph.

Figure 14.6.2: Weight of a subgraph.



Weight of blue subgraph:
3 + (-2) + (-1) + 5.2 + 4 = 9.2

The goal is to span the vertices of the graph while minimizing the total weight of the edges used:

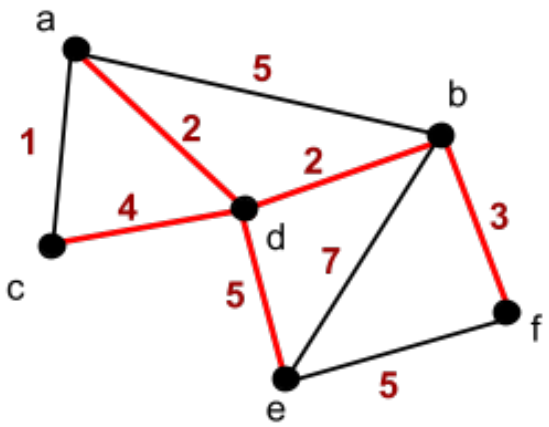
Definition 14.6.2: Minimum spanning tree.

A ***minimum spanning tree*** (MST) of a weighted graph, is a spanning tree T of G whose weight is no larger than any other spanning tree of G.

A graph can have more than one minimum spanning tree because there can be more than one spanning tree with the same weight. In the extreme case, if the weight of every edge is 1, then every spanning tree has weight n - 1 and every spanning tree is also a minimum spanning tree.

PARTICIPATION ACTIVITY 14.6.1: Minimum spanning trees.

The graph below shows a weighted graph and a spanning tree highlighted in red.



1) What is the weight of the highlighted spanning tree?

Check Show answer

2) Is the highlighted spanning tree a minimum weight spanning tree? Type: Yes or No

Check Show answer

Prim's algorithm for minimum spanning tree

We present here a classic algorithm for finding minimum spanning trees developed by mathematician Robert Prim in 1957. The algorithm starts with a single vertex and grows the minimum spanning tree by adding a vertex and an edge in each step. If T is the partial tree at a particular point in the algorithm's execution, the set of candidate edges to be added next are the edges with one endpoint inside T and one endpoint outside T. Prim's algorithm selects the smallest weight edge among all the candidate edges and adds the selected edge to T along with the endpoint of the edge that is not in T. The animation below illustrates:

PARTICIPATION ACTIVITY 14.6.2: Prim's algorithm for finding an MST.

Animation content:

undefined

Animation captions:

1. Start with T consisting of a single initial vertex. The edges incident to the initial vertex are the eligible edges.
2. Select the eligible edge with min weight. Add the selected edge and its endpoint to T.
3. The eligible edges have one endpoint in T and one outside of T. Edges become eligible if one endpoint is T's new vertex and the other is outside T.
4. T now has 2 vertices. Select the min eligible edge. Add the selected edge and its endpoint to T. T now consists of 3 vertices.
5. Update the eligible edges and select the min eligible edge from the new set of eligible edges. Add the selected edge and its endpoint to T. T has 4 vertices.
6. After the next iteration, T has 5 vertices. In updating the set of eligible edges, an edge is removed because both endpoints of the edges are now in T.
7. In each remaining iteration, a new vertex and edge are added to T.
8. The Minimum Spanning Tree is complete when all the vertices are in T.

Here is the pseudo-code for Prim's algorithm:

Figure 14.6.3: Prim's algorithm for finding and MST.

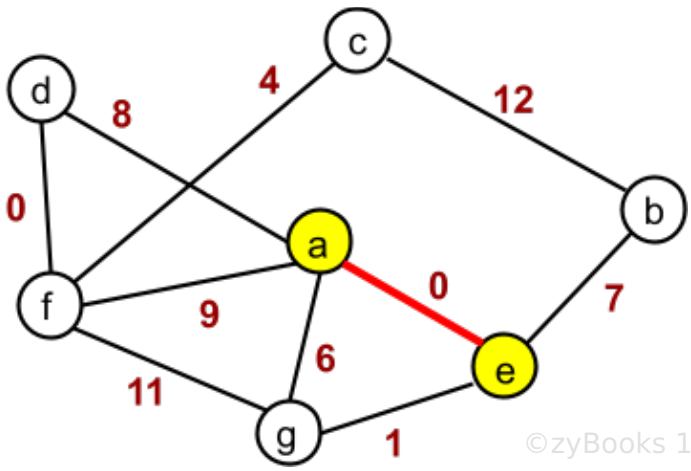
Input: An undirected, connected, weighted graph G.
Output: T, a minimum spanning tree for G.

T := Ø.
Pick any vertex in G and add it to T.

For j = 1 to n-1
    Let C be the set of edges with one endpoint inside T and one endpoint outside T.
    Let e be a minimum weight edge in C.
    Add e to T.
    Add the endpoint of e not already in T to T.
End-for

PARTICIPATION ACTIVITY 14.6.3: Prim's algorithm.

Consider Prim's algorithm applied to the graph below. Vertices a and e have already been added to the MST T.



1) Which set corresponds to the set of eligible edges?

- {a, g}, {a, d}, {a, f}, {e, b}, {f, g}
- {a, g}, {a, d}, {a, f}, {e,b}
- {a, g}, {e, g}, {a, d}, {a, f}, {e, b}

2) Which edge is added next?

- {a, g}
- {e,g}
- {f, g}

3) After vertex g is added, which edge becomes ineligible?

- {a, g}
- {a, f}
- {d, f}

4) After vertex g is added, which edge becomes eligible?

- {a, f}
- {g, f}
- {d, f}

It is important to establish formally that the algorithm actually does finish with a minimum spanning tree of the input graph.

Theorem 14.6.1: Prim's algorithm finds a minimum spanning tree.

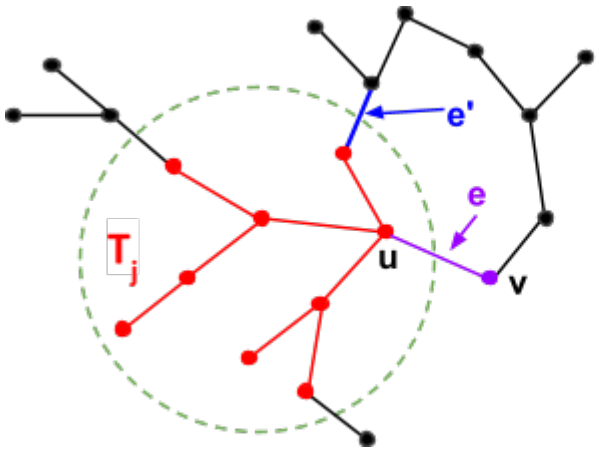
Prim's algorithm finds a minimum spanning tree of the input weighted graph.

Proof 14.6.1: Prim's algorithm finds a minimum spanning tree.

Proof.
Prim's algorithm incrementally builds a spanning tree. Define T_j to be the partial tree after j edges have been added. T_0 is the initial vertex and T_{n-1} is the final spanning tree. We prove by induction on j that T_j is contained within a minimum spanning tree T. Since T_{n-1} is a spanning tree, it must also then be a minimum spanning tree.

The base case is T_0 which is a single vertex. A minimum spanning tree contains all the vertices, so it must also contain T_0.

Now suppose that T_j is contained in a minimum spanning tree called T. Let e = {u, v} be the next edge added to get T_{j+1}. Let u be the endpoint of e inside T_j and v be the endpoint of e outside T_j. If e is also in T, then T_{j+1} is also contained in the MST T and the induction step of the proof is complete. Suppose instead that e is not in T. If e is added to T, then T ∪ {e} has n edges, so the resulting graph can not be a tree and must contain a cycle. Since the cycle involves the edge e, there is a path in T that connects v to u that does not use edge e. The path must cross from outside T_j (from v) to inside T_j (to u) and therefore there is an edge e' in the cycle with one endpoint outside of T_j and one in T_j.



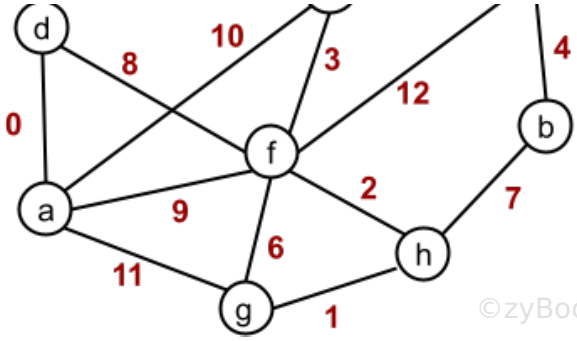
T has e' and not e

T' has e and not e

The edge e' was a candidate edge at the time e was chosen, so w(e') ≥ w(e). Construct a new spanning tree T' by taking e' out of T and putting in e. The result is a spanning tree T' whose weight is no more than T and also contains T_{j+1}. Therefore T' is an MST that contains T_{j+1}. ■

PARTICIPATION ACTIVITY 14.6.4: Prim's algorithm for MST.

Consider an execution of Prim's algorithm applied to the weighted graph below, starting from vertex a.



Order the edges according to the order in which Prim's algorithm adds them to the MST. (Note that not all the edges are added).

If unable to drag and drop, refresh the page.

{a, d} **{f, e}** **{c, b}** **{d, f}** **{h, g}** **{e, c}** **{f, h}**

1

2

3

4

5

6

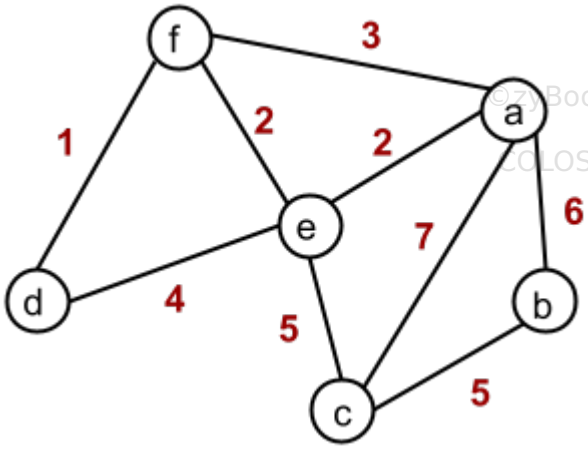
7

Reset

Additional exercises

EXERCISE | 14.6.1: Minimum spanning trees that include a given edge.

An undirected weighted graph G is given below:



(a) Use Prim's algorithm to compute the minimum spanning tree for the weighted graph. Start the algorithm at vertex a. Show the order in which the edges are added to the tree.

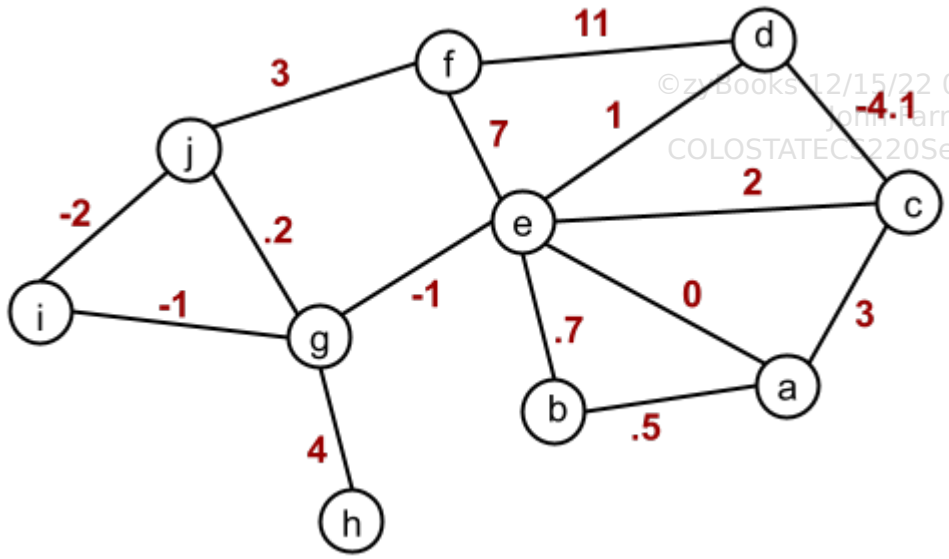
(b) What is the minimum weight spanning tree for the weighted graph in the previous question subject to the condition that edge {d,e} is in the spanning tree?

(c) How would you generalize this idea? Suppose you are given a graph G and a particular edge {u,v} in the graph. How would you alter Prim's algorithm to find the minimum spanning tree subject to the condition that {u,v} is in the tree?

©zyBooks 12/15/22 00:31 1361995
John Farrell
COLOSTATECS220SeaboltFall2022

EXERCISE 14.6.2: Adapting an MST after an update to an edge weight.

An undirected weighted graph G is given below:



(a) Use Prim's algorithm to compute the minimum spanning tree for the weighted graph. Start the algorithm at vertex a. Show the order in which the edges are added to the tree.

(b) Now suppose that the weight of edge {e,f} is decreased from 7 to -3. Is there a way to find the minimum spanning tree of the altered graph without running Prim's algorithm from scratch?

EXERCISE 14.6.3: A common edge in all MSTs.

(a) Do two different minimum spanning trees of a weighted undirected graph always have an edge in common?

EXERCISE 14.6.4: Distinct edge weights and unique MSTs.

(a) Prove that if there is no pair of edges that have the same weight, then the minimum spanning tree is unique.

14.7 Depth First and Breadth First Graph

Traversal

dfbf traverses a graph breadth first and computes distances from root to reachable nodes; it then traverses the graph depth firsthand determines whether the graph from root is cyclic.

422102.2723990.qx3zqy7

LAB ACTIVITY | 14.7.1: Depth First and Breadth First Graph Traversal | 0 / 100

Downloadable files

`dfbf.py` , `in3.txt` , `in2.txt` , and `in1.txt` **Download**

dfbf.py Load default template...

```
1  '''
2  Breadth First and Depth First Search
3
4  The objective is to write a Python program that traverses graphs in BFS
5  and DFS manner. BFS will determine the shortest path distance (number of
6  edges) from the root for each node reachable from the root. DFS will find
7  cycles in the graph of nodes reachable from the root. Study the lecture on
8  graphs, in particular graph traversals.
9
10
11 Some helper code is provided. Don't change it. Don't change your main,
12 it is used to check your code's correctness.
13
14 It is your job to implement dfs and bfs. In both dfs and bfs, visit
15 children of a node in left to right order, i.e., if adj is the
16 adjacency list of a node, visit the children as follows: for nxt in adj
17
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run command

`python3 dfbf.py` Additional arguments

Run program Input (from above) → **dfbf.py** (Your program) → O

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```


©zyBooks 12/15/22 00:31 1361995
John Farrell
COLOSTATECS220SeaboltFall2022