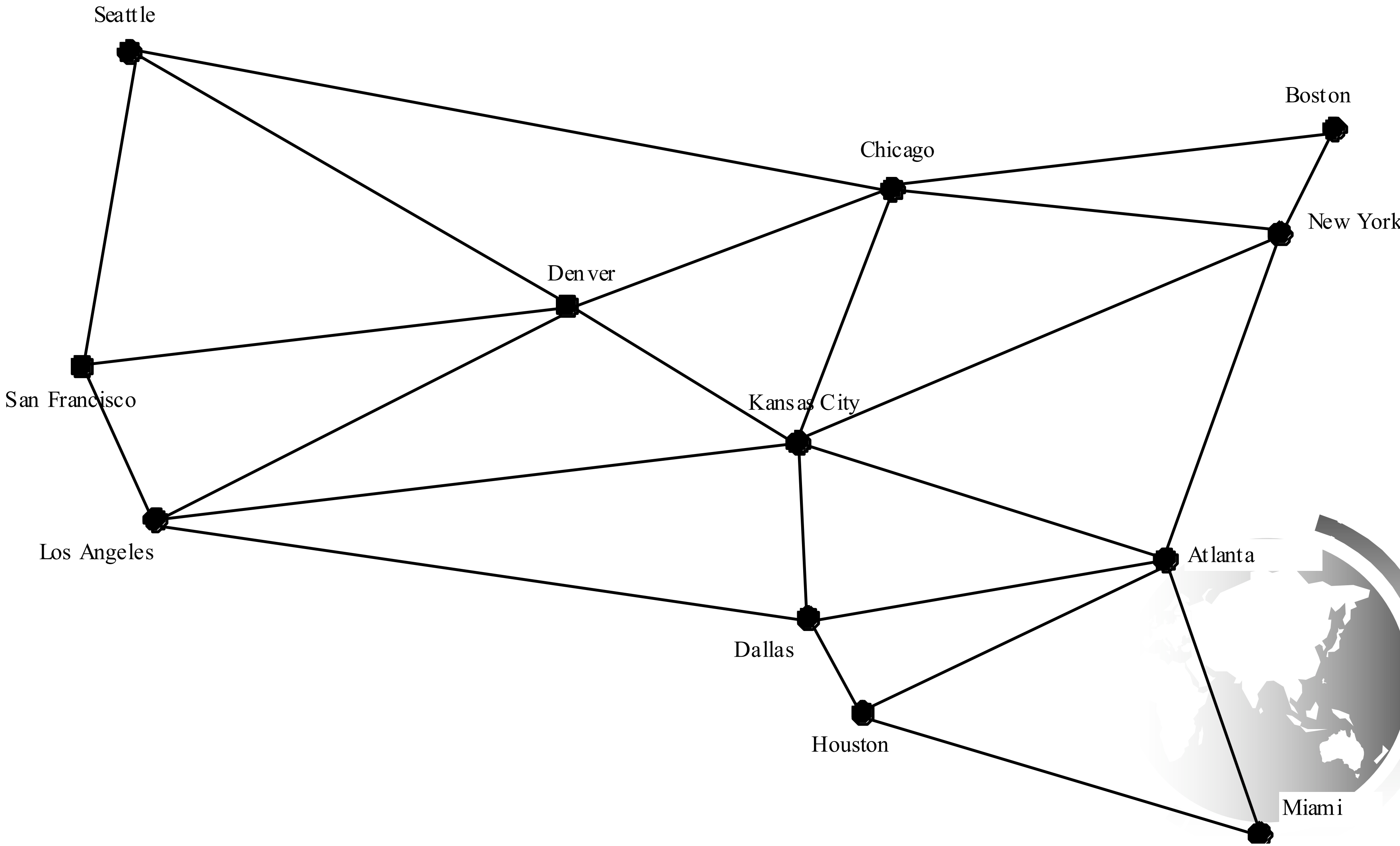
Chapter 22 Graphs and Applications

Objectives

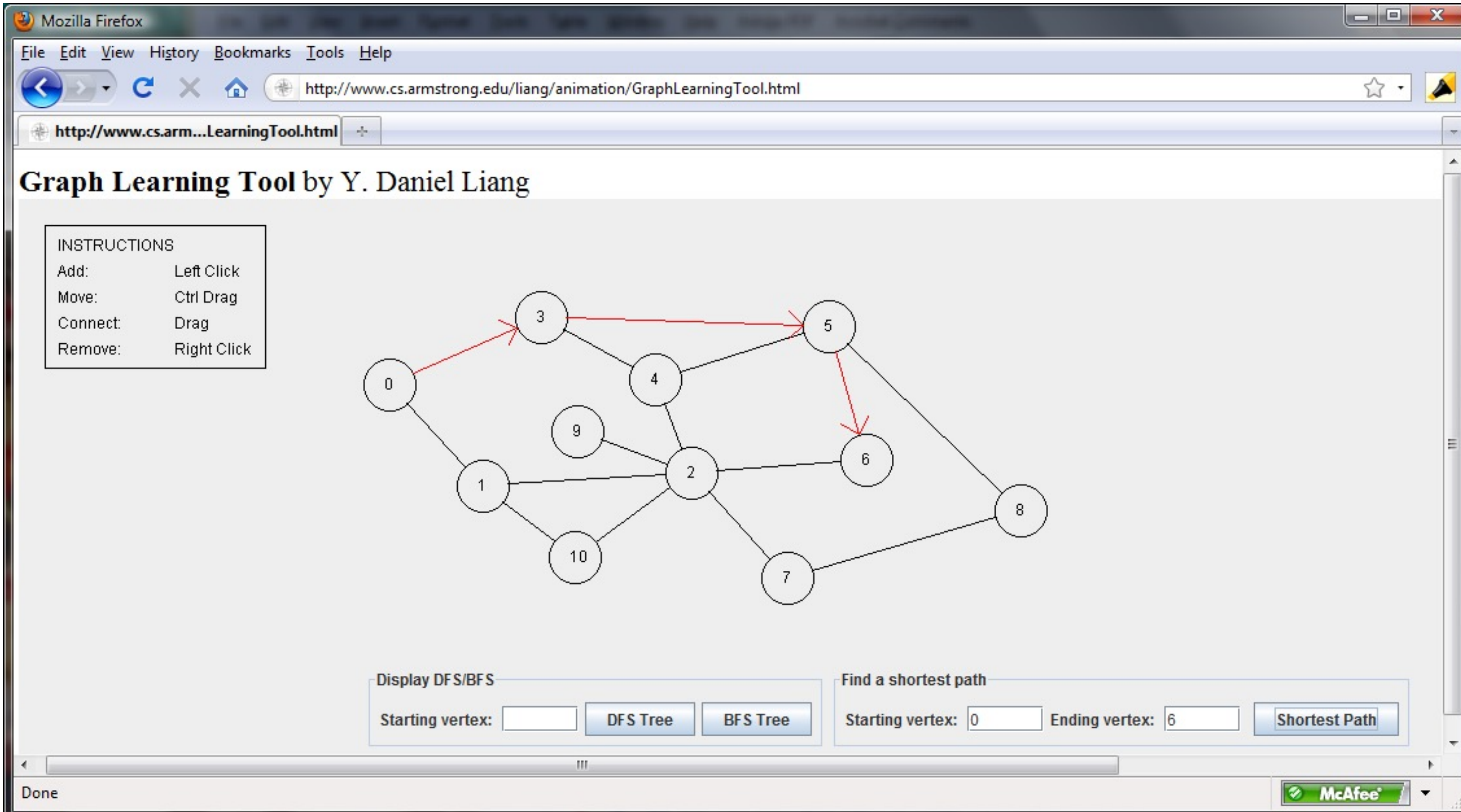
- To model real-world problems using graphs and explain the Seven Bridges of Königsberg problem (§22.1).
- To describe the graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§22.2).
- To represent vertices and edges using lists, adjacent matrices, and adjacent lists (§22.3).
- To model graphs using the Graph class (§22.4).
- To display graphs visually (§22.5).
- To represent the traversal of a graph using the Tree class (§22.6).
- To design and implement depth-first search (§22.7).
- To solve the connected-circle problem using depth-first search (§22.8).
- To design and implement breadth-first search (§22.9).
- To solve the nine-tail problem using breadth-first search (§22.10).

Modeling Using Graphs

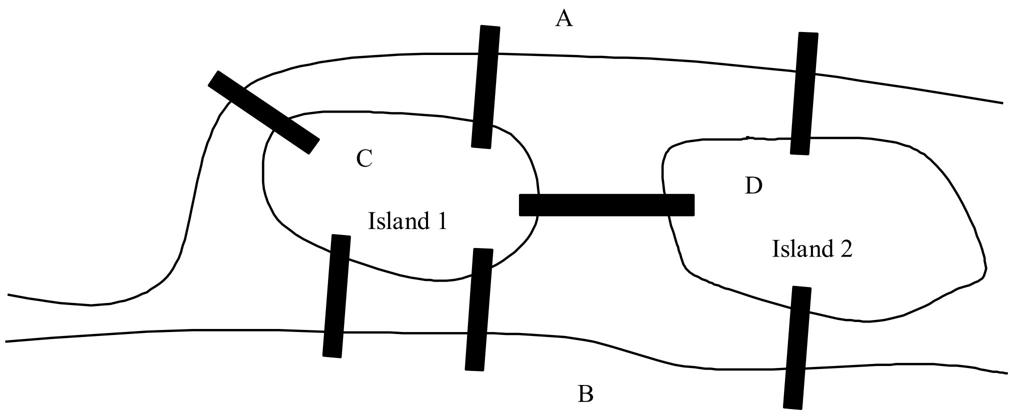


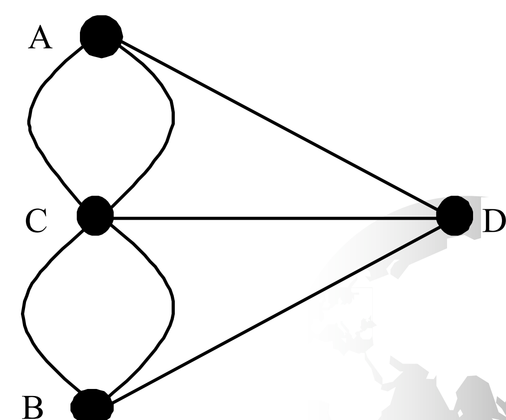
Graph Animation

https://liveexample.pearsoncmg.com/dsanimation/GraphLearningTooleBook.html



Seven Bridges of Königsberg





Basic Graph Terminologies

What is a graph?

Define a graph

Directed vs. undirected graphs

Weighted vs. unweighted graphs

Adjacent vertices

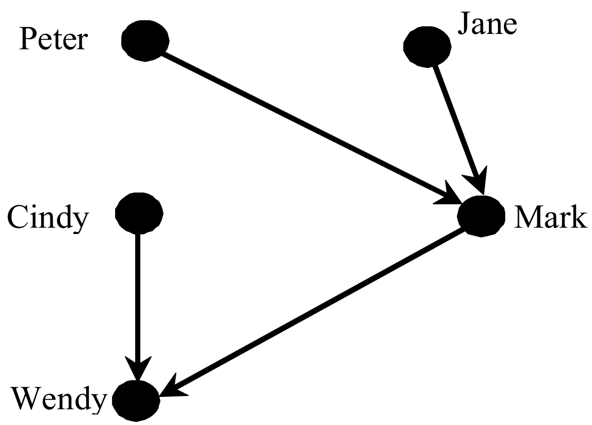
Incident

Degree

Neighbor

loop

Directed vs Undirected Graph



Basic Graph Terminologies

Parallel edge

Simple graph

Complete graph

Spanning tree

Representing Graphs

Representing Vertices

Representing Edges: Edge Array

Representing Edges: Edge Objects

Representing Edges: Adjacency Matrices

Representing Edges: Adjacency Lists

Representing Vertices

V = ["Seattle", "San Francisco", "Los Angeles", "Denver", "Kansas City", "Chicago", "Boston", "New York", "Atlanta", "Miami", "Dallas", "Houston"]

Representing Edges: Edge List

```
edges = [
    [0, 1], [0, 3], [0, 5],
    [1, 0], [1, 2], [1, 3],
    [2, 1], [2, 3], [2, 4], [2, 10],
    [3, 0], [3, 1], [3, 2], [3, 4], [3, 5],
    [4, 2], [4, 3], [4, 5], [4, 7], [4, 8], [4, 10],
    [5, 0], [5, 3], [5, 4], [5, 6], [5, 7],
    [6, 5], [6, 7],
    [7, 4], [7, 5], [7, 6], [7, 8],
    [8, 4], [8, 7], [8, 9], [8, 10], [8, 11],
    [9, 8], [9, 11],
    [10, 2], [10, 4], [10, 8], [10, 11],
    [11, 8], [11, 9], [11, 10]
  ]
```

Representing Edges: Edge Object

```
class Edge:
    def __init__(self, u, v):
        self.u = u
        self.v = v


edgeList = []
edgeList.append(Edge(0, 1))
edgeList.append(Edge(0, 3))
edgeList.append(Edge(0, 5))
...
```

Representing Edges: Adjacency Matrix

```
adjacencyMatrix = [
    [0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0], # Seattle
    [1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0], # San Francisco
    [0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0], # Los Angeles
    [1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # Denver
    [0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0], # Kansas City
    [1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0], # Chicago
    [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # Boston
    [0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0], # New York
    [0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1], # Atlanta
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1], # Miami
    [0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1], # Dallas
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0]  # Houston
]
```

Representing Edges: Adjacency List

neighbors = [[1, 3, 5], [0, 2, 3], [1, 3, 4, 10], [0, 1, 2, 4, 5],
[2, 3, 4, 5, 7, 8, 10], [0, 3, 4, 6, 7], [5, 7],
[4, 5, 6, 8], [4, 7, 9, 10, 11], [8, 11],
[2, 4, 8, 11], [8, 9, 10]]

Seattle	neighbors[0]	[1, 3, 5]
San Francisco	neighbors[1]	[0, 2, 3]
Los Angeles	neighbors[2]	[1, 3, 4, 10]
Denver	neighbors[3]	[0, 1, 2, 4, 5]
Kansas City	neighbors[4]	[2, 3, 4, 5, 7, 8, 10]
Chicago	neighbors[5]	[0, 3, 4, 6, 7]
Boston	neighbors[6]	[5, 7]
New York	neighbors[7]	[4, 5, 6, 8]
Atlanta	neighbors[8]	[4, 7, 9, 10, 11]
Miami	neighbors[9]	[8, 11]
Dallas	neighbors[10]	[2, 4, 8, 11]
Houston	neighbors[11]	[8, 9, 10]

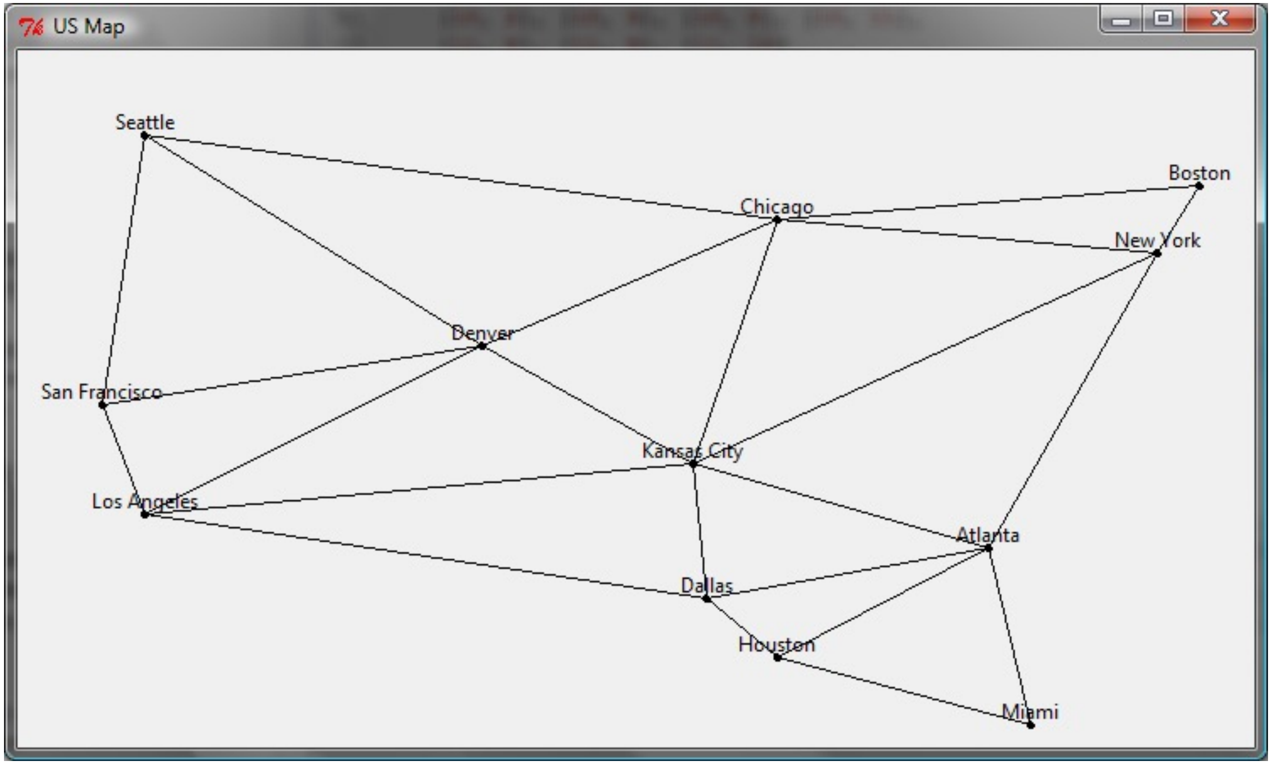
Modeling Graphs

Graph	
vertices: list	Stores vertices.
neighbors: list of adjacency list	Stores edges.
Graph(vertexList: list, edgeList: list)	Constructs a graph with the specified vertices and edges.
getAdjacencyList(edgeList: list): list	Returns an adjacency list from edgeList.
getSize(): int	Returns the number of vertices in the graph.
getVertices(): list	Returns the vertices in the graph.
getVertex(index): object	Returns the vertex at the specified index.
getIndex(v: object): int	Returns the index for the specified vertex.
getNeighbors(index: int): list	Returns the neighbors of vertex with the specified index.
getDegree(v: object): int	Returns the degree for a specified vertex.
printEdges(): None	Prints the edges.
clear(): None	Clears the graph.
addVertex(v: object): None	Adds a vertex to the graph.
addEdge(u: object, v: object): None	Adds an edge (from u to v) to the graph.
+dfs(index: int): Tree	Obtains a depth-first search tree.
+bfs(index: int): Tree	Obtains a breadth-first search tree.

Graph

TestGraph

Graph Visualization



Displayable

Displayable

GraphView

Graph Traversals

Depth-first search and breadth-first search

Both traversals result in a spanning tree, which can be modeled using a class.

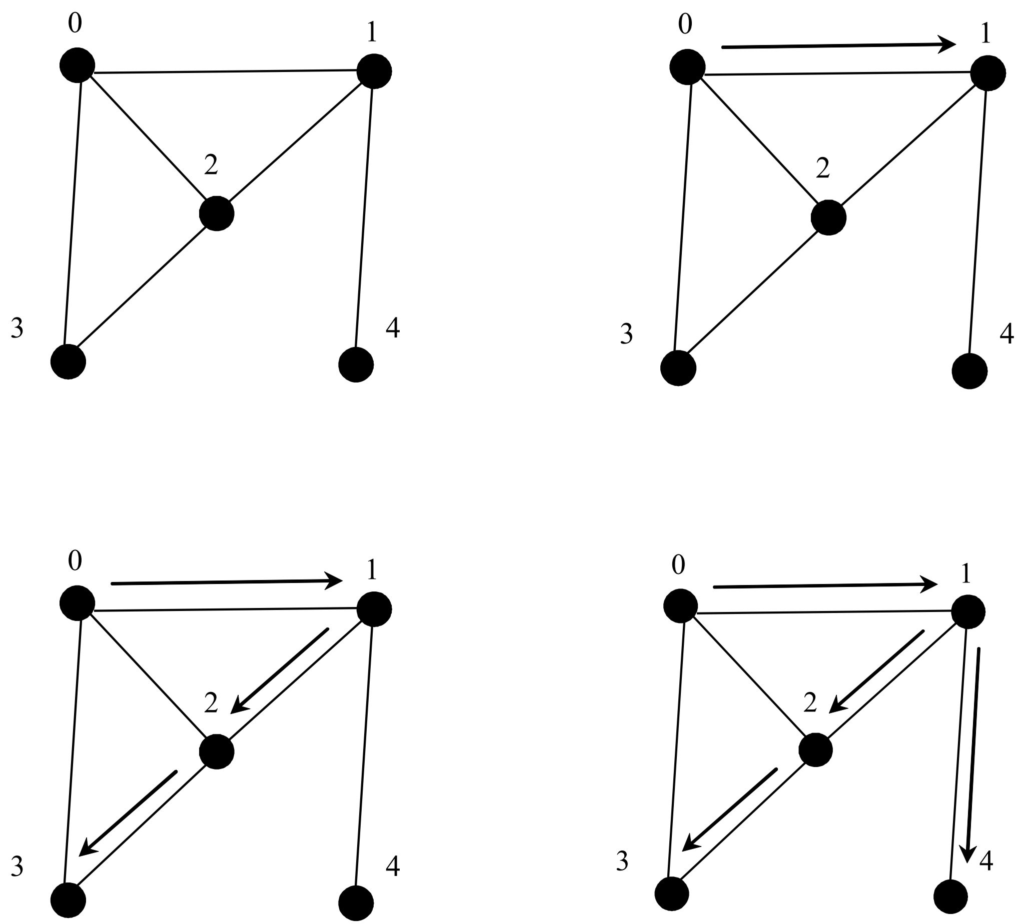
Tree	
root: int	The root of the tree.
parent: list	The parents of the vertices in index.
searchOrders: list	The orders for traversing the vertices in index.
vertices: list	The vertices of the graph.
Tree(root: int, parent: list, searchOrders: list, vertices: list)	Constructs a tree with the specified root, parent, search orders, and vertices for the graph.
getRoot(): int	Returns the root of the tree.
getSearchOrders(): list	Returns the order of vertices searched.
getParent(index: int): int	Returns the parent for the specified vertex index.
getNumberOfVerticesFound(): int	Returns the number of vertices searched.
getPath(index: int): list	Returns a list of vertices from the specified vertex index to the root.
printPath(index: int): None	Displays a path from the root to the specified vertex.
printTree(): None	Displays tree with the root and all edges.

Depth-First Search

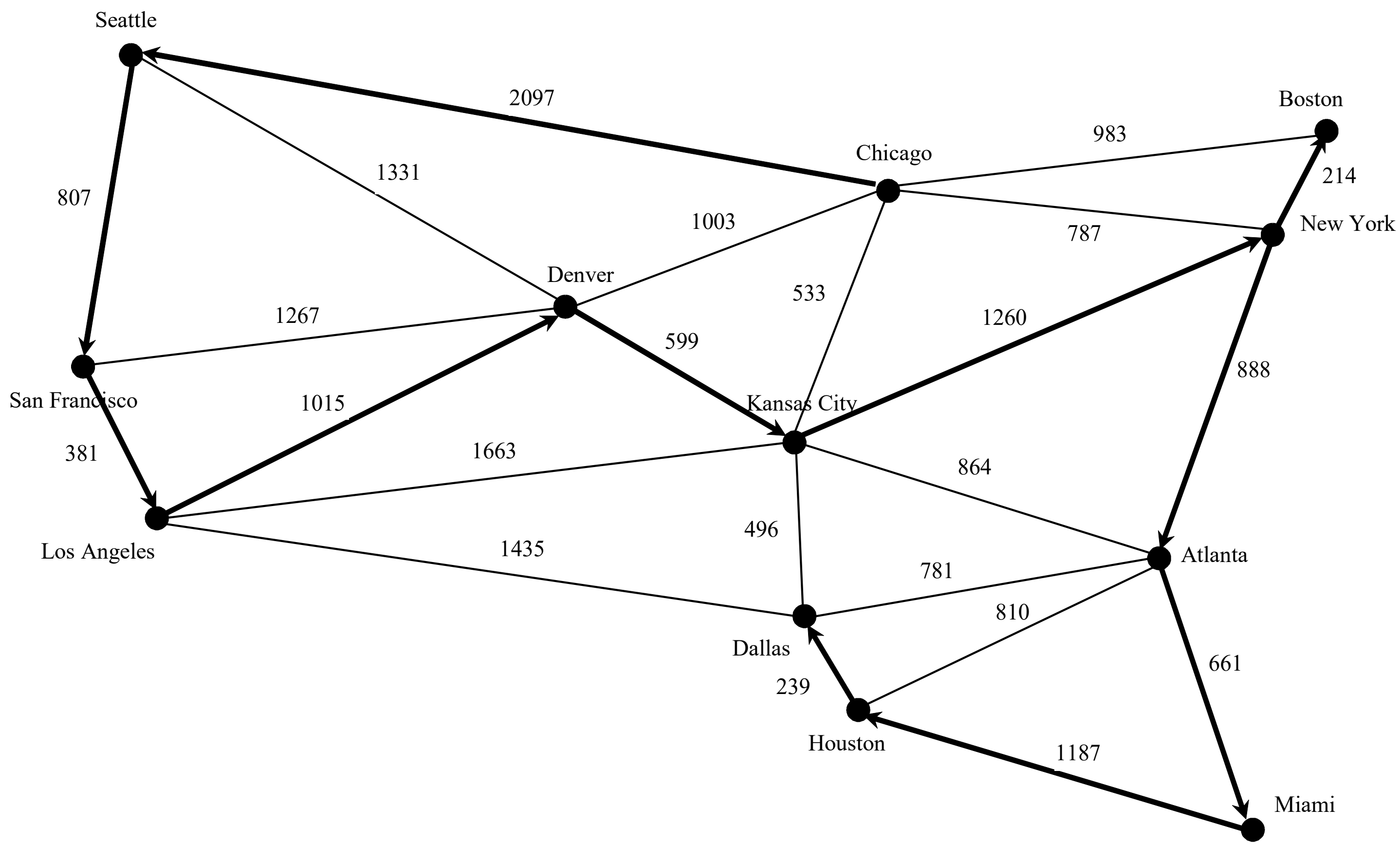
The depth-first search of a graph is like the depth-first search of a tree discussed in §19.2.3, "Tree Traversal." In the case of a tree, the search starts from the root. In a graph, the search can start from any vertex.

```
dfs(vertex v):
    visit v
    for each neighbor w of v:
        if w has not been visited:
            dfs(w)
```

Depth-First Search Example



Depth-First Search Example



TestDFS

Applications of the DFS

Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected (See Exercise 22.2.)

Detecting whether there is a path between two vertices. (See Exercise 22.3)

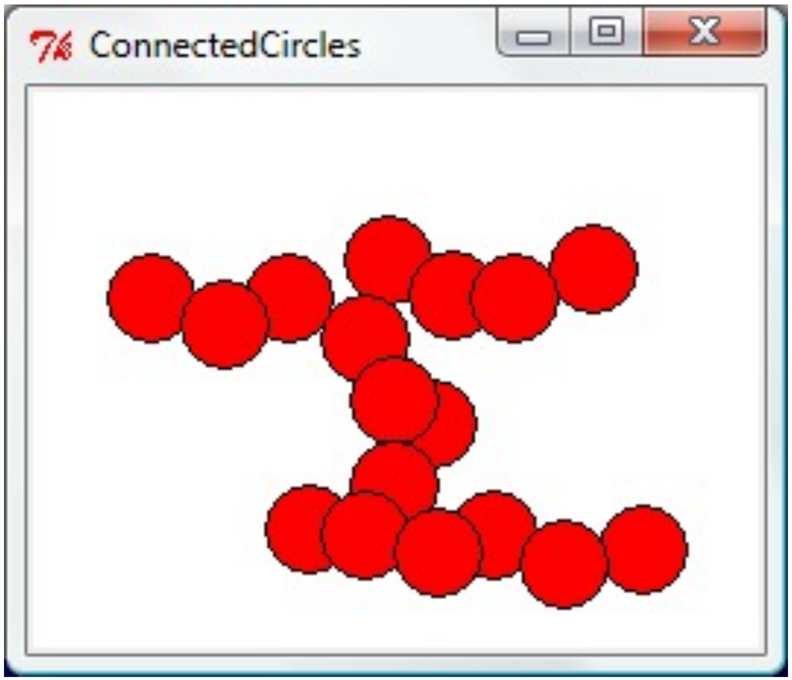
Finding a path between two vertices. (See Exercise 22.3)

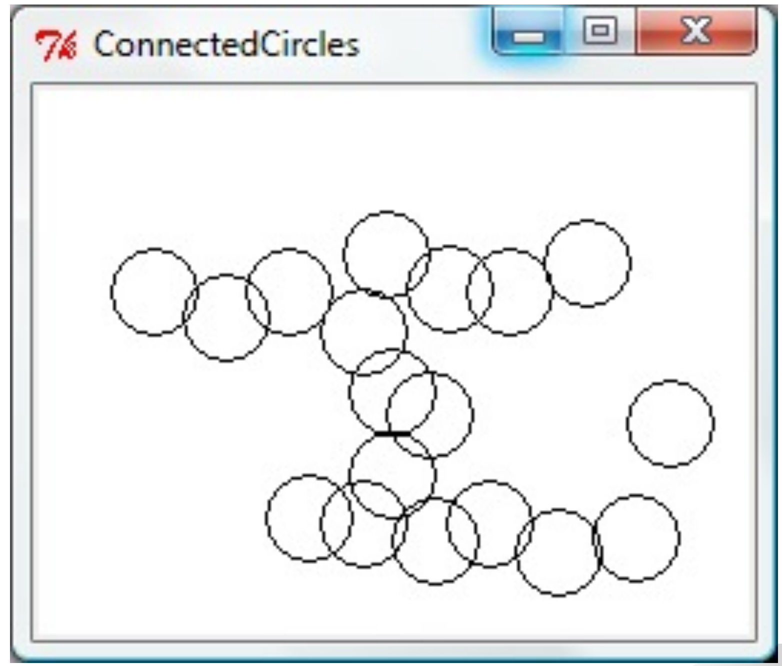
Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path. (See Exercise 22.2)

Detecting whether there is a cycle in the graph. (See Exercise 22.4)

Finding a cycle in the graph. (See Exercise 22.5)

The Connected Circles Problem





ConnectedCircles

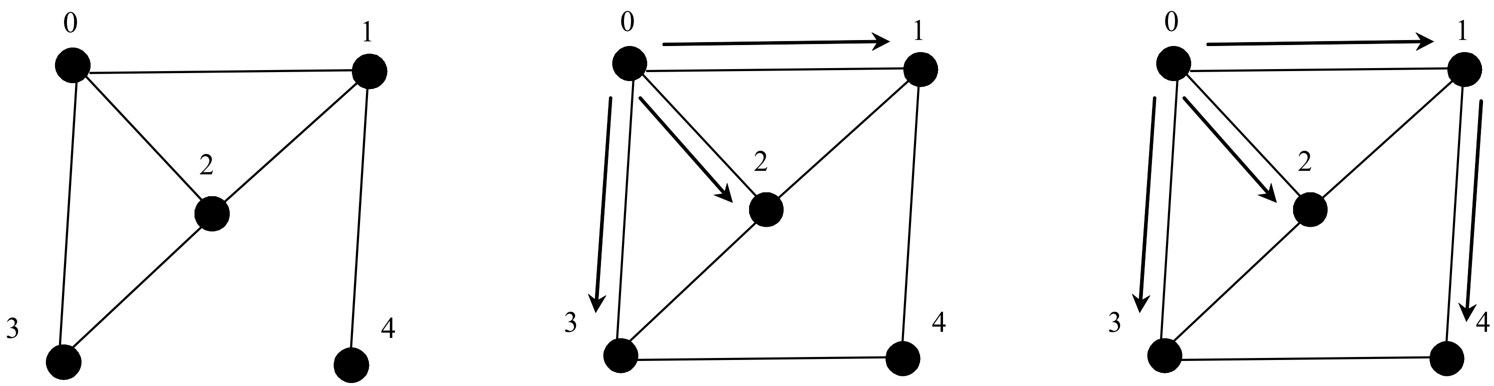
Breadth-First Search

The breadth-first traversal of a graph is like the breadth-first traversal of a tree discussed in §22.2.3, “Tree Traversal.” With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root from left to right, and so on.

Breadth-First Search Algorithm

```
bfs(vertex v):
    create an empty queue for storing vertices to be visited
    add v into the queue
    mark v visited
    while the queue is not empty:
        dequeue a vertex, say u, from the queue
        add u into a list of traversed vertices
        for each neighbor w of u
            if w has not been visited:
                add w into the queue
                mark w visited
```

Breadth-First Search Example



Queue: 0

isVisited[0] = True

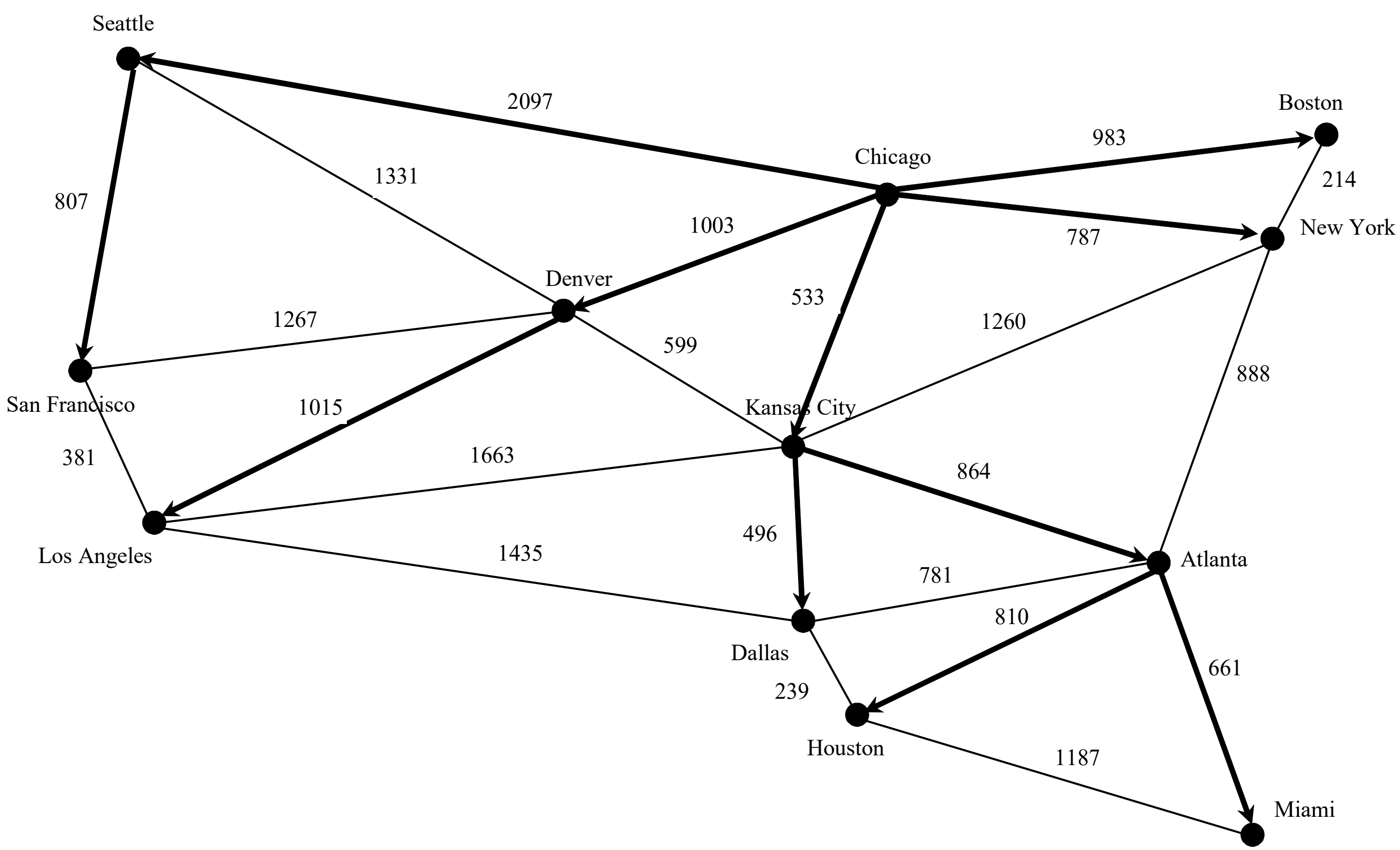
Queue: 1 2 3

isVisited[1] = True, isVisited[2] = True, isVisited[3] = True

Queue: 2 3 4

isVisited[4] = True

Breadth-First Search Example



TestBFS

Applications of the BFS

Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.

Detecting whether there is a path between two vertices.

Finding a shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node (see Review Question 22.10).

Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.

Detecting whether there is a cycle in the graph. (See Exercise 22.4)
Finding a cycle in the graph. (See Exercise 22.5)

Testing whether a graph is bipartite. A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set. (See Exercise 22.8)

The Nine Tail Problem

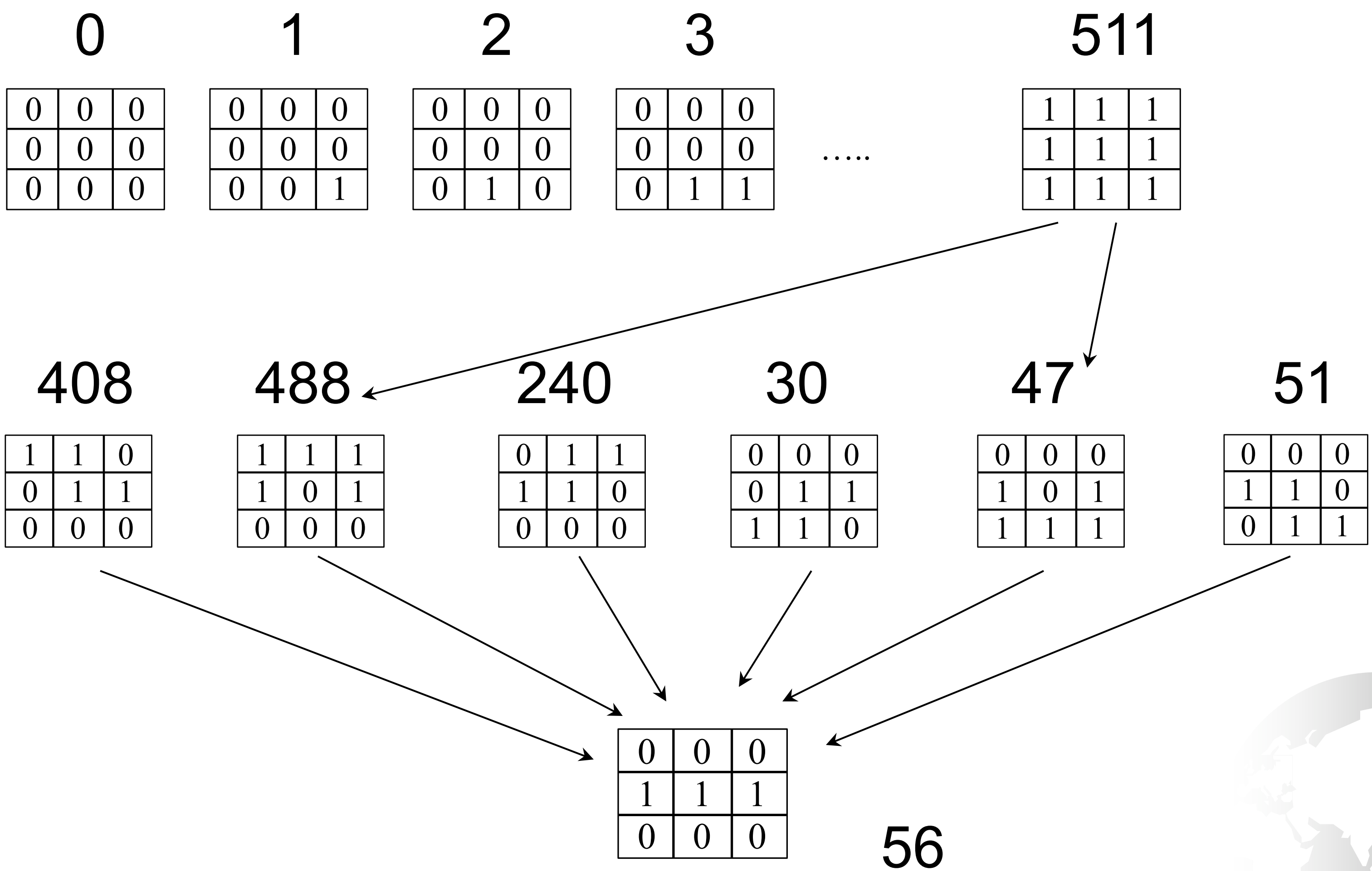
The problem is stated as follows. Nine coins are placed in a three by three matrix with some face up and some face down. A legal move is to take any coin that is face up and reverse it, together with the coins adjacent to it (this does not include coins that are diagonally adjacent). Your task is to find the minimum number of the moves that lead to all coins face down.

H	H	H
T	T	T
H	H	H

H	H	H
T	H	T
T	T	T

T	T	T
T	T	T
T	T	T

Model the Nine Tail Problem



NineTailModel

NineTailModel	
tree: Tree	A tree rooted at node 511.
NineTailModel()	Constructs a model for the nine tail problem and obtains the tree.
getShortestPath(nodeIndex: int): list	Returns a path from the specified node to the root. The path returned consists of the node labels in a list.
getEdges(): list	Returns an edge list for the graph.
<u>getNode(index: int): list</u>	Returns a node consisting of nine characters of H's and T's.
<u>getIndex(node: list): int</u>	Returns the index of the specified node.
getFlippedNode(node: list, position: int): int	Flips the node at the specified position and returns the index of the flipped node.
flipACell(node: list, row: int, column: int): None	Flips the node at the specified row and column.
<u>printNode(node: list): None</u>	Displays the node to the console.

NineTailModel NineTail