```
N = [
    ('g', 'b',),
    ('g', 'd',),
    ('g', 'f',),
    ('b', 'd',),
    ('b', 'f',),
    ('d', 'f',)
]

G = Graph(N)

G.show()
```

Layout options include: “acyclic”, “circular”, “ranked”, “graphviz”, “planar”, “spring”, or “tree”.

A planar graph can be drawn without any crossing edges. The default graph layout does not ensure the planar layout of a planar graph. Sage will return an error if you try to plot a non-planar graph with the planar layout.

```
G.plot(layout='planar').show()
```

Sage’s `planar` algorithm sets the vertex positions. Alternatively, we can specify the positions in a dictionary. Let’s position the G node in the center.

```
positions = {
    'g': (0, 0),
    'd': (-1, 1),
    'b': (1, 1),
    'f': (0, -1)
}

G.plot(pos=positions).show()
```

The following graph modeling the intervals in the C major scale is challenging to read. Let’s think about how we can improve the presentation.

```
I = [
    ("c", "d", "M2"), ("c", "e", "M3"), ("c", "f", "P4"),
        ("c", "g", "P5"), ("c", "a", "M6"), ("c", "b", "M7"),
    ("d", "e", "M2"), ("d", "f", "m3"), ("d", "g", "P4"),
        ("d", "a", "P5"), ("d", "b", "M6"), ("d", "c", "m7"),
    ("e", "f", "m2"), ("e", "g", "m3"), ("e", "a", "P4"),
        ("e", "b", "P5"), ("e", "c", "m6"), ("e", "d", "m7"),
    ("f", "g", "M2"), ("f", "a", "M3"), ("f", "b", "a4"),
        ("f", "c", "P5"), ("f", "d", "M6"), ("f", "e", "M7"),
    ("g", "a", "M2"), ("g", "b", "M3"), ("g", "c", "P4"),
        ("g", "d", "P5"), ("g", "e", "M6"), ("g", "f", "m7"),
    ("a", "b", "M2"), ("a", "c", "m3"), ("a", "d", "P4"),
        ("a", "e", "P5"), ("a", "f", "m6"), ("a", "g", "m7"),
    ("b", "c", "m2"), ("b", "d", "m3"), ("b", "e", "P4"),
        ("b", "f", "d5"), ("b", "g", "m6"), ("b", "a", "m7"),
]

C = DiGraph(I, multiedges=True,)

C.plot(edge_labels=True).show()
```

In this case, the graph is not planar. The `circular` layout organizes the

vertices for improved readability.

```
C.plot(edge_labels=True, layout='circular')
```

7.2.5 View in a New Tab

Increasing the `figsize` improves the definition of the arrows. For an even better view of the Graph, right-click the image and view it in a new tab.

```
C.plot(
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```

7.2.6 Edge Style

The options for `edge_style` include “`solid`”, “`dashed`”, “`dotted`”, or "`dashdot`”.

```
C.plot(
    edge_style='dashed',
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```

Improve the definition between the edges by using a different color for each edge. The `color_by_label` method automatically maps the colors to edges.

```
C.plot(
    edge_style='dashed',
    color_by_label=True,
    edge_labels=True,
    layout='circular',
    figsize=30
).show()
```

7.2.7 3-Dimensional

View a 3D representation of graph with `show3d()`. Click and drag the image to change the perspective. Zoom in on the image by pinching your computer's touchpad.

```
G = graphs.CubeGraph(3)
G.show3d()
```

```
G = graphs.TetrahedralGraph()
G.show3d()
```

```
G = graphs.IcosahedralGraph()
G.show3d()
```

```
G = graphs.DodecahedralGraph()
G.show3d()
```

```
G = graphs.CompleteGraph(5)
G.show3d()
```

7.3 Paths

A path between two vertices u and v is a sequence of consecutive edges starting at u and ending at v.

To get all paths between two vertices:

```
G = Graph({1: [2, 3], 2: [3], 3: [4]})
G.show()
G.all_paths(1, 4)
```

The length of a path is defined as the number of edges that make up the path.

Finding the shortest path between two vertices can be achieved using the `shortest_path()` function:

```
G = Graph({1: [2, 3], 2: [3], 3: [4]})
G.show()
G.shortest_path(1, 4)
```

A graph is said to be connected if there is a path between any two vertices in the graph.

To determine if a graph is connected, we can use the `is_connected()` function:

```
G = Graph({1: [2, 3], 2: [3], 3: [4]})
G.show()
G.is_connected()
```

A connected component of a graph G is a maximal connected subgraph of G. If the graph G is connected, then it has only one connected component.

For example, the following graph is not connected:

```
G = Graph({1: [2, 3], 2: [4], 5: [6]})
G.show()
G.is_connected()
```

To identify all connected components of a graph, the `connected_components()` function can be utilized:

```
G = Graph({1: [2, 3], 2: [4], 5: [6]})
G.show()
G.connected_components()
```

We can visualize the graph as a disjoint union of its connected components, by plotting it.

```
G = Graph({1: [2, 3], 2: [4], 5: [6, 7], 6: [7]})
G.show()
```

The diameter of a graph is the length of the longest path among all of a graph's shortest paths between any two vertices.

```
G = Graph({1: [2, 3], 2: [3, 4], 3: [4]})
G.show()

# Calculates the diameter of the graph.
G.diameter()
```

A graph is bipartite if its set of vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set:

```
G = Graph({1: [2, 3], 2: [4], 3: [4]})
G.show()
G.is_bipartite()
```

7.4 Isomorphism

Informally, we can say that an **isomorphism** is a relation of sameness between graphs. Let's say that the graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $f : V \to V'$ such that $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$.

This means there is a bijection between the set of vertices such that every time two vertices determine an edge in the first graph, the image of these vertices by the bijection also determines an edge in the second graph, and vice versa. Essentially, the two graphs have the same structure, but the vertices are labeled differently.

Notes. Graph isomorphism identifies structures relevant to chemistry, biology, machine learning, and neural networks.

```
C = Graph(
  {
    'a': ['b', 'c', 'g'],
    'b': ['a', 'd', 'h'],
    'c': ['a', 'd', 'e'],
    'd': ['b', 'c', 'f'],
    'e': ['c', 'f', 'g'],
    'f': ['d', 'e', 'h'],
    'g': ['a', 'e', 'h'],
    'h': ['b', 'f', 'g']
  }
)

D = Graph(
  {
    1: [2, 6, 8],
    2: [1, 3, 5],
    3: [2, 4, 8],
    4: [3, 5, 7],
    5: [2, 4, 6],
    6: [1, 5, 7],
    7: [4, 6, 8],
    8: [1, 3, 7]
  }
)

C.show()
D.show()
```

The sage `is_isomorphic()` method can be used to check if two graphs are isomorphic. The method returns `True` if the graphs are isomorphic and `False` if the graphs are not isomorphic.

```
C.is_isomorphic(D)
```

The **invariants under isomorphism** are conditions that can be checked to determine if two graphs are not isomorphic. If one of these fails then the graphs are not isomorphic. If all of these are true then the graph may or may not be isomorphic. The three conditions for invariants under isomorphism are:

$$G = (V, E) \text{ is connected if and only if } G' = (V', E') \text{ is connected}$$

$$|V| = |V'| \text{ and } |E| = |E'|$$

$$\text{degree sequence of } G = \text{ degree sequence of } G'$$

To summarize, if one graph is connected and the other is not, then the graphs are not isomorphic. If the number of vertices and edges are different, then the graphs are not isomorphic. If the degree sequences are different, then the graphs are not isomorphic. If all three invariants are satisfied, then the graphs may or may not be isomorphic.

Let's define a function to check if two graphs satisfy the invariants under isomorphism. Make sure you run the next cell to define the function before using the function.

```
def invariant_under_isomorphism(G1, G2):
    print("Are␣both␣graphs␣connected?␣", end="")
    are_connected: bool = (
        G1.is_connected() == G2.is_connected()
    )
    print("Yes" if are_connected else "No")

    print(
        "Do␣both␣graphs␣have␣same␣number␣of␣"
        "vertices␣and␣edges?␣", end=""
    )
    have_equal_vertex_and_edge_counts: bool = (
        G1.order() == G2.order() and
        G1.size() == G2.size()
    )
    print(
        "Yes" if have_equal_vertex_and_edge_counts else "No"
    )

    # Sort the degree-sequences because
    # the order of vertices doesn't matter.
    print(
        "Do␣both␣graphs␣have␣the␣same␣degree␣sequence?␣",
        end=""
    )
    have_same_degree_sequence: bool = (
        sorted(G1.degree_sequence()) ==
        sorted(G2.degree_sequence())
    )
    print("Yes" if have_same_degree_sequence else "No")

    # All checks
    are_invariant_under_isomorphism =(
```

```
        are_connected and
        have_equal_vertex_and_edge_counts and
        have_same_degree_sequence
    )
    print(
        "\nTherefore, the graphs {0} isomorphic.".format(
            "may be" if are_invariant_under_isomorphism
            else "are not"
        )
    )
```

If we use `invariant_under_isomorphism` on the C and D, the output will let's know that the graphs may or may not be isomorphic. We can use the `is_isomorphic()` method to check if the graphs are definitively isomorphic.

```
invariant_under_isomorphism(C, D)
```

Let's construct a different pair of graphs A and B defined as follow

```
A = Graph(
  [
    ('a', 'b'),
    ('b', 'c'),
    ('c', 'f'),
    ('f', 'd'),
    ('d', 'e'),
    ('e', 'a')
  ]
)

B = Graph(
  [
    (1, 5),
    (1, 9),
    (5, 9),
    (4, 6),
    (4, 7),
    (6, 7)
  ]
)

A.show()
B.show()
```

This time, if we apply `invariant_under_isomorphism` function on A and B, the output will show us that they are not isomorphic.

```
invariant_under_isomorphism(A, B)
```

7.5 Euler and Hamilton

7.5.1 Euler

An **Euler path** is a path that uses every edge of a graph exactly once. An Euler path that is a circuit is called an **Euler circuit**.

The idea of an Euler path emerged from the study of the **Königsberg bridges** problem. Leonhard Euler wanted to know if it was possible to walk

through the city of Königsberg, crossing each of its seven bridges exactly once. This problem can be modeled as a graph, with the land masses as vertices and the bridges as edges.

```
konigsberg = [('A', 'B', 'b_1'),
              ('A', 'B', 'b_2'),
              ('A', 'C', 'b_3'),
              ('A', 'C', 'b_4'),
              ('D', 'A', 'b_5'),
              ('D', 'B', 'b_6'),
              ('D', 'C', 'b_7')]
G = Graph(konigsberg, multiedges=True)
G.show(edge_labels=True)
```

Notes. Eulerian circuits and paths have practical applications for reducing travel and costs in logistics, waste management, the airline industry, and postal service.

While exploring this problem, Euler discovered the following:

- A connected graph has an **Euler circuit** if and only if every vertex has an even degree.
- A connected graph has an **Euler path** if and only if there are at most two vertices with an odd degree.

We say that a graph is **Eulerian** if contains an Euler circuit.

We can use Sage to determine if a graph is Eulerian.

```
G.is_eulerian()
```

Since this returns `False`, we know that the graph is not Eulerian. Therefore, it is not possible to walk through the city of Königsberg, crossing each of its seven bridges exactly once.

We can use `path=True` to determine if a graph contains an Euler path. Sage will return the beginning and the end of the path.

```
G = Graph([(1, 2), (2, 3), (3, 4), (4, 1), (2, 4), (1, 3),
    (1, 4)], multiedges=True)
G.show()
G.is_eulerian(path=True)
```

If the graph is Eulerian, we can ask Sage to find an Euler circuit with the `eulerian_circuit` function. Let's take a look at the following graph.

```
G = Graph([(1, 2), (2, 3), (2, 3),(3, 4), (4, 1), (2, 4),
    (1, 3), (1, 4)], multiedges=True)
G.show()
G.eulerian_circuit()
```

If we are not interested in the edge labels, we can set `labels=False`. We can also set `return_vertices=True` to get a list of vertices for the path

```
G = graphs.CycleGraph(6)
G.eulerian_circuit(labels=False, return_vertices=True)
```

7.5.2 Hamilton

A **Hamilton path** is a path that uses every vertex of a graph exactly once. A Hamilton path that is a circuit is called a **Hamilton circuit**. If a graph contains a Hamilton circuit, we say that the graph is **Hamiltonian**.

Hamilton created the "Around the World" puzzle. The object of the puzzle was to start at a city and travel along the edges of the dodecahedron, visiting all of the other cities exactly once, and returning back to the starting city.

We can represent the dodecahedron as a graph and use Sage to determine if it is Hamiltonian. See for yourself if the dodecahedron is Hamiltonian.

```
graphs.DodecahedralGraph().show()
```

We can ask Sage to determine if the dodecahedron is Hamiltonian.

```
graphs.DodecahedralGraph().is_hamiltonian()
```

By running `Graph.is_hamiltonian??` we see that Sage uses the `traveling_salesman_problem()` function to determine if a graph is Hamiltonian.

The traveling salesperson problem is a classic optimization problem. Given a list of cities and the lengths between each pair of cities, what is the shortest possible route that visits each city and returns to the original city? This is one of the most difficult problems in computer science. It is **NP-hard**, meaning that no efficient algorithm is known to solve it. The complexity of the problem increases with the number of nodes. When working with many nodes, the algorithm can take a long time to run.

Let's explore the following graph:

```
G = Graph({1:{3:2, 2:1, 4:3, 5:1}, 2:{3:6, 4:3, 5:1},
    3:{4:5, 5:3}, 4:{5:5}})
G.show(edge_labels=True)
```

We can ask Sage if the graph contains a Hamiltonian cycle.

```
G.hamiltonian_cycle(algorithm='backtrack')
```

The function `hamiltonian_cycle` returns `True` and lists an example of a Hamiltonian cycle as the list of vertices `[1, 2, 3, 4, 5]`. This is just one of the many Hamiltonian cycles that exist in the graph. Now lets find the minimum Hamiltonian cycle.

```
h = G.traveling_salesman_problem(use_edge_labels=True,
    maximize=False)
h.show(edge_labels=True)
```

Now we have the plot of the minimum Hamiltonian cycle. The minimum Hamiltonian cycle is the shortest possible route that visits each city and returns to the original city. The minimum Hamiltonian cycle is the solution to the traveling salesperson problem. We can ask Sage for the sum of the weights of the edges in the minimum Hamiltonian cycle.

```
sumWeights = sum(h.edge_labels())
print(sumWeights)
```

If there is no Hamiltonian cycle, Sage will return `False`. If we use the `backtrack` algorithm, Sage will return a list that represents the longest path found.

```
G = Graph([(1, 2), (1, 3), (2, 3), (1,4), (4, 7), (3, 5),
    (5, 8), (8, 9), (2,6), (6, 9), (7, 9)])
G.show()
G.hamiltonian_cycle(algorithm='backtrack')
```

7.6 Graphs in Action

Imagine you are a bike courier tasked with making deliveries to each City Colleges of Chicago (CCC) campus location. Per your contract, you get paid per delivery, not per hour. Therefore, finding the most efficient delivery route is in your best interest. We assume the bike delivery routes are the same distance in each direction.

7.6.1 Bike Courier Delivery Route Problem

Let's make a plan to solve our delivery route problem.

1. Find the distances in miles between each CCC location.

2. Make a graph of the CCC locations. Each location is a node. Each edge is a bike route. The weight of the edges represents the distance of the bike route between locations.

3. Use the traveling salesperson algorithm to calculate the optimal delivery route.

7.6.2 Locations

Table 7.6.1 CCC Addresses

Name	Address
Harold Washington College	30 E. Lake Street, Chicago, IL 60601
Harry Truman College	1145 West Wilson Ave, Chicago, IL 60640
Kennedy-King College	6301 South Halsted St, Chicago, IL 60621
Malcolm X College	1900 W. Jackson, Chicago, IL 60612
Olive-Harvey College	10001 South Woodlawn Ave, Chicago, IL 60628
Richard J. Daley College	7500 South Pulaski Rd, Chicago, IL 60652
Wilbur Wright College	4300 N. Narragansett Ave, Chicago, IL 60634

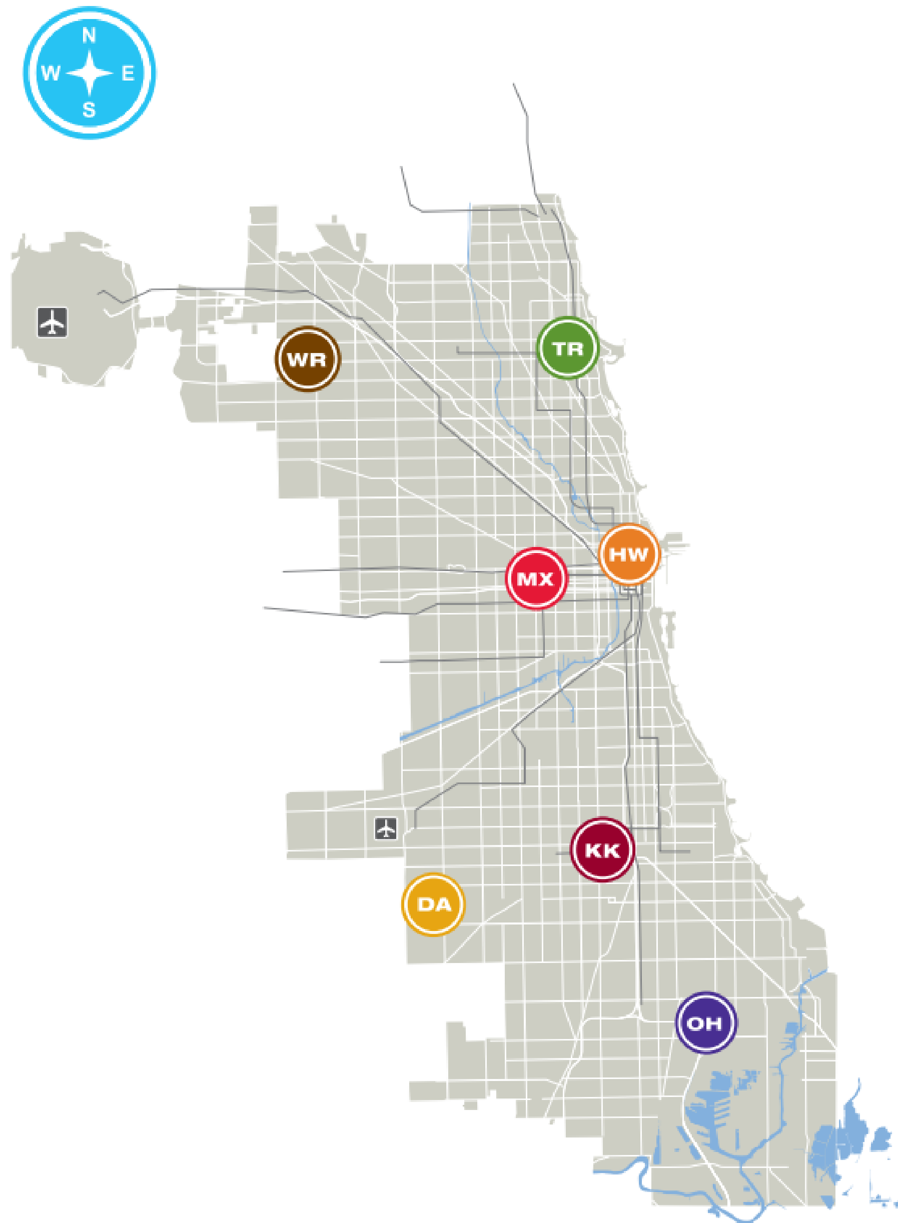

Figure 7.6.2 City Colleges of Chicago

7.6.3 Graph

We will represent each College as a node with the initials of the College name. The weight of the edge will represent the miles in between the locations. Since we are using bike routes, we are assuming each direction between two locations has the same distance. For example, express the route between Harold Washington College and Harry Truman College as `("HW", "HT", 6.5)`.

```
routes = [
    ("HW", "HT", 6.5),
    ("HW", "KK", 8.3),
    ("HW", "MX", 3.2),
```

```
    ("HW", "OH", 15.4),
    ("HW", "RD", 11.9),
    ("HW", "WW", 10.7),

    ("HT", "KK", 13.6),
    ("HT", "MX", 7.1),
    ("HT", "OH", 22.1),
    ("HT", "RD", 17.3),
    ("HT", "WW", 7.9),

    ("KK", "MX", 8.3),
    ("KK", "OH", 8.1),
    ("KK", "RD", 5.7),
    ("KK", "WW", 16.9),

    ("MX", "OH", 16.2),
    ("MX", "RD", 10.2),
    ("MX", "WW", 10.2),

    ("OH", "RD", 10.0),
    ("OH", "WW", 24.9),

    ("RD", "WW", 18.3)
]
routes
```

Create a `Graph` from the edge `list`:

```
G = Graph(routes)
G.show(edge_labels=True)
```

The trailing zeros of the floating point values are hard to read. Let's loop through the edge `list` and display the numbers with 3 points of precision.

```
for u, v, label in G.edge_iterator():
    G.set_edge_label(u, v, n(label, digits=3))

G.show(edge_labels=True)
```

Since this graph is not planar, improve the layout with the `"circular"` parameter. We can also improve the readability by increasing the `vertex_size` and `figsize`.

```
G.show(
    edge_labels=True,
    layout="circular",
    vertex_size=500,
    figsize=10,
)
```

Now that we have a clearer idea of the routes, let's find the most efficient delivery route using the traveling salesperson algorithm.

```
optimal_route =
   G.traveling_salesman_problem(use_edge_labels=True,
   maximize=False)
optimal_route.show(
    edge_labels=True,
```

```
    vertex_size=500,
)
```

We can set the vertex positions to resemble their positions on the map. We can use the latitude and longitude values of the locations and then reverse them when we supply the values to the position `dictionary`.

```
positions = {
    'HW': (-87.62682604591349, 41.88609733324964),
    'HT': (-87.65901943241516, 41.9646769664519),
    'KK': (-87.6435785385309, 41.77847328856264),
    'MX': (-87.67453475017268, 41.87800548491064),
    'OH': (-87.5886722734757, 41.71006715754713),
    'RD': (-87.72315805813204, 41.75677704810169),
    'WW': (-87.78738482318016, 41.95836512405638),
}

Graph(optimal_route).show(
    pos=positions,
    edge_labels=True,
    vertex_size=500,
    figsize=10,
)
```

Chapter 8

Trees

This chapter completes the preceding one by explaining how to ask Sage to decide whether a given graph is a tree and then introduce further searching algorithms for trees.

8.1 Definitions and Theorems

Given a graph, a **cycle** is a circuit with no repeated edges. A **tree** is a connected graph with no cycles. A graph with no cycles and not necessarily connected is called a **forest**.

Let $G = (M, E)$ be a graph. The following are all equivalent:

- G is a tree.
- For each pair of distinct vertices, there exists a unique path between them.
- G is connected, and if $e \in E$ then the graph $(V, E - e)$ is disconnected.
- G contains no cycles, but by adding one edge, you create a cycle.
- G is connected and $|E| = |v| - 1$.

Let's explore the following graph:

```
data = {
  1: [4],
  2: [3, 4, 5],
  3: [2],
  4: [1, 2, 6, 7],
  5: [2, 8],
  6: [4, 9, 11],
  7: [4],
  8: [5, 10],
  9: [6],
  10: [8],
  11: [6]
  }

G = Graph(data)
G.show()
```

Notes. Trees are a common data structure used in file explorers, parsers, and decision making.

Let's ask Sage if this graph is a tree.

```
G.is_tree()
```

If we remove an edge, we can see that the graph is no longer a tree.

```
G_removed_edge = G.copy()
G_removed_edge.delete_edge((1, 4))
G_removed_edge.show()
G_removed_edge.is_tree()
```

However, we can see that the graph is still a forest.

```
G_removed_edge.is_forest()
```

If we add an edge, we can see that the graph contains a cycle and is no longer a tree.

```
G_added_edge = G.copy()
G_added_edge.add_edge((1, 2))
G_added_edge.show()
G_added_edge.is_tree()
```

8.2 Search Algorithms

The graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq \{\{u, v\} \in E \mid u, v \in V'\}$.

The subgraph $G' = (V', E')$ is a **spanning subgraph** of $G = (V, E)$ if $V' = V$.

A **spanning tree** for the graph G is a spanning subgraph of G that is a tree.

Given a graph, various algorithms can calculate a spanning tree, including depth-first search and breadth-first search.

Breadth-first search algorithm

1. Choose a vertex of the graph (root) arbitrarily.
2. Travel all the edges incident with the root vertex.
3. Give an order to this set of new vertices added.
4. Consider each of these vertices as a root, in order, and add all the unvisited incident edges that do not produce a cycle.
5. Repeat the method with the new set of vertices.
6. Follow the same procedure until all the vertices are visited.

The output of this algorithm is a spanning tree.

The `breadth_first_search()` function provides a flexible method for traversing both directed and undirected graphs. Let's consider the following graph:

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4},
    "c":{"e":2}, "d":{"a":5, "c":4}})
G.show()
print(list(G.breadth_first_search(start="a",
```

```
    report_distance=True)))
```

In the example above, the `start` parameter begins the traversal at vertex `a`. The `report_distance=True`, parameter reports pairs in the format `(vertex, distance)`. Distance is the length of the path from the start vertex. From the output above, we see:

- The distance from vertex `a` to vertex `a` is `0`.
- The distance from vertex `a` to vertex `d` is `1`.
- The distance from vertex `a` to vertex `e` is `1`.
- The distance from vertex `a` to vertex `c` is `1`.
- The distance from vertex `a` to vertex `b` is `2`.

We can also set the parameter `edges=True` to return the edges of the BFS tree. Sage will raise an error if you use the `edges` and `report_distance` parameters simultaneously.

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4},
    "c":{"e":2}, "d":{"a":5, "c":4}})
s = list(G.breadth_first_search("a", edges=True))
print(s)
Graph(s)
```

The above graph is a spanning tree, but not necessarily a minimum spanning tree. Let's check how many spanning trees exist.

```
G.spanning_trees_count()
```

Iterate over all the spanning trees of a graph with `spanning_trees()`.

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4},
    "c":{"e":2}, "d":{"a":5, "c":4}})
spanning_trees = list(G.spanning_trees(labels=True))
for i, tree in enumerate(spanning_trees):
    print(f"Spanning␣Tree␣{i␣+␣1}:␣{tree.edges()}")
    show(tree.plot())
```

Given a weighted graph of all possible spanning trees we can calculate, we may be interested in the minimal one. A **minimal spanning tree** is a spanning tree whose sum of weights is minimal. Prim's Algorithm calculates a minimal spanning tree.

Prim's Algorithm: Keep two disjoint sets of vertices. One (L) contains vertices that are in the growing spanning tree, and the other (R) that are not in the growing spanning tree.

1. Choose a vertex $u \in V$ arbitrarily. At this step, $L = \{u\}$ and $R = V - \{u\}$.
2. In R, select the cheapest vertex connected to the growing spanning tree L and add it to L
3. Follow the same procedure until all the vertices are in L

The output of this algorithm is a minimal spanning tree.

```
G = Graph({"a":{"c":8, "e":1}, "b":{"c":6, "d":4},
    "c":{"e":2}, "d":{"a":5, "c":4}})
G.show(edge_labels = True)
```

We can ask Sage for the minimal spanning tree of this graph. By running `Graph.min_spanning_tree??` We can see that `min_spanning_tree()` uses a variation of Prim's Algorithm by default. We can also use other algorithms such as Kruskal, Boruvka, or NetworkX.

Notes. Minimal spanning trees influence the efficient design of networks and roads.

```
G.min_spanning_tree(by_weight=True)
```

From the output of `min_spanning_tree(by_weight=True)`, we see an edge `list` of the minimal spanning tree. Each element of the edge lis is a `tuple` where the first two values are vertices, and the third value is the edge weight or label.

Let's visualize the minimal spanning tree.

```
h = Graph(G.min_spanning_tree(by_weight=True))
h.show(edge_labels = True)
```

Let's define a function to view the minimal spanning tree in the context of the original graph. The function parameters include:

- `graph`: A Sage Graph object.
- `mst_color`: Color for edges part of the MST (default: `'darkred'`).
- `non_mst_color`: Color for edges not part of the MST (default: `'lightblue'`).
- `figsize`: Dimensions for the graph image.

```
def visualize_mst(input_graph, mst_color='darkred',
    non_mst_color='lightblue', figsize=None):
    try:
        if not input_graph.is_connected():
            print("The graph must be connected")
            return

        mst_edges =
            input_graph.min_spanning_tree(by_weight=True)
        print("MST Edges:", mst_edges)
        Graph(mst_edges).show(edge_labels=True,
            figsize=figsize, edge_color=mst_color)

        edge_colors = {mst_color: [], non_mst_color: []}

        mst_edge_set = set((v1, v2) for v1, v2, _ in
            mst_edges)

        for edge in input_graph.edges():
            v1, v2, _ = edge
            if (v1, v2) in mst_edge_set or (v2, v1) in
                mst_edge_set:
                edge_colors[mst_color].append((v1, v2))
            else:
                edge_colors[non_mst_color].append((v1, v2))
```

```
        print("MST␣overlaid␣on␣the␣original␣graph:")
        p = input_graph.plot(edge_labels=True,
            edge_colors=edge_colors, figsize=figsize)
        show(p)

    except Exception as e:
        print("Error:", e)
```

Let's generate a random graph and view the minimal spanning tree.

```
import random

vertices = 5
G = Graph([(i, j, random.randint(1, 20)) for i in
    range(vertices) for j in range(i+1, vertices)])

visualize_mst(G)
```

The following graph contains 9 vertices.

```
import random

vertices = 9
G = Graph([(i, j, random.randint(1, 20)) for i in
    range(vertices) for j in range(i+1, vertices)])

visualize_mst(G, figsize=10)
```

The following graph contains 15 vertices.

```
import random

vertices = 15
G = Graph([(i, j, random.randint(1, 20)) for i in
    range(vertices) for j in range(i+1, vertices)])

visualize_mst(G, figsize=10)
```

8.3 Trees in Action

Imagine your task is to create a railway between all the City Colleges of Chicago (CCC) campus locations. The contract requests that you use minimal track material to save construction costs. For simplicity's sake, assume each railway is a straight line between campuses.

8.3.1 Railway Problem

Let's make a plan to solve our railway construction optimization problem.

1. Find the latitude and longitude of each CCC campus location.
2. Use the Haversine formula to calculate the distances between the locations. The Haversine formula requires latitude and longitude for inputs and computes the shortest path between two points on a sphere.

3. Make a graph of the CCC campuses. Each location is a node. Each railway path is an edge. Each railway path is the shortest path between locations. The weight of the edges represents the distance between locations.

4. Find the minimum spanning tree (MST) of the CCC graph.

8.3.2 Location Distances

Table 8.3.1 City Colleges of Chicago Locations

Name	**(Latitude, Longitude)**
Harold Washington College	(41.88609733324964, -87.62682604591349)
Harry Truman College	(41.9646769664519, -87.65901943241516)
Kennedy-King College	(41.77847328856264, -87.6435785385309)
Malcolm X College	(41.87800548491064, -87.67453475017268)
Olive-Harvey College	(41.71006715754713, -87.5886722734757)
Richard J. Daley College	(41.75677704810169, -87.72315805813204)
Wilbur Wright College	(41.95836512405638, -87.78738482318016)

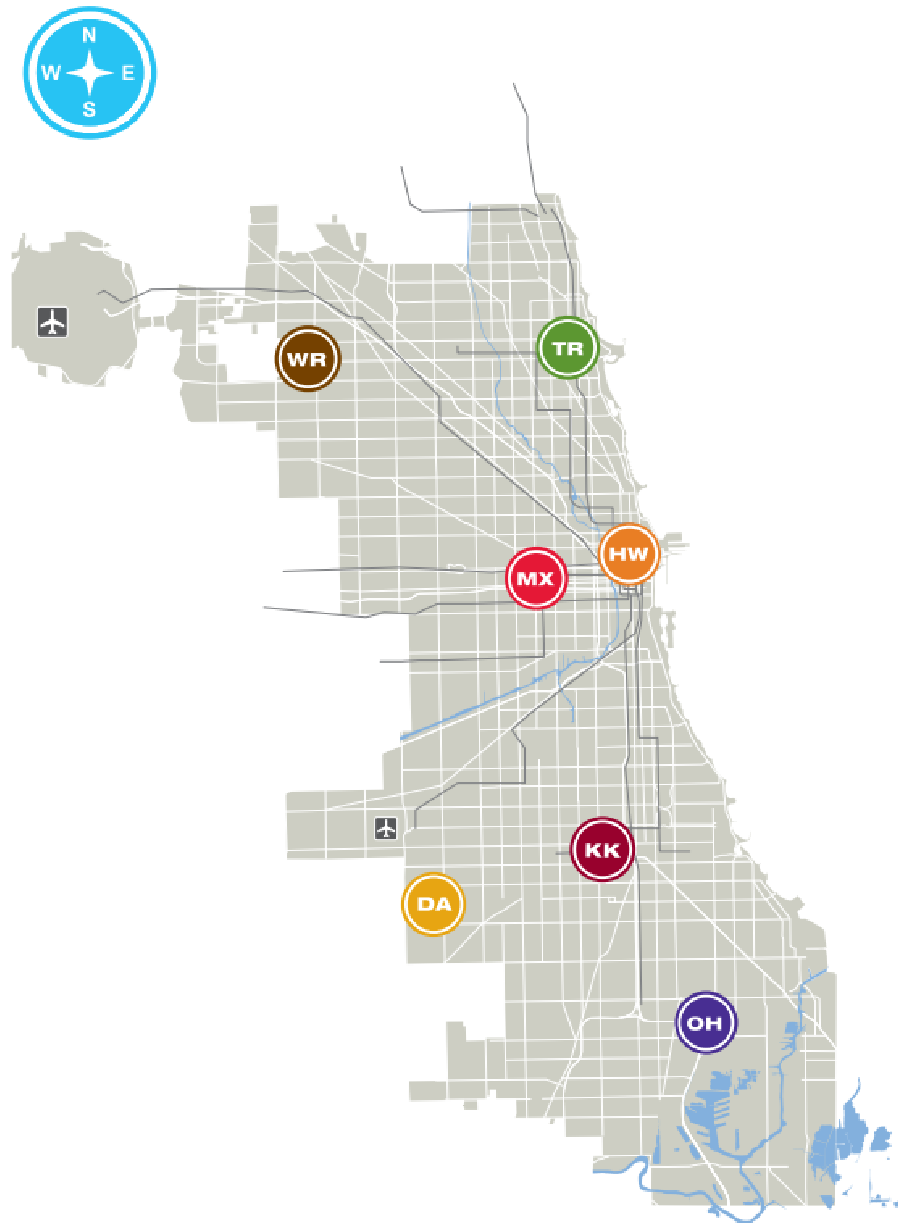

Figure 8.3.2 City Colleges of Chicago

Now, let's calculate the distances between campus locations. We will first create a `dictionary` to store the campus name, latitude, and longitude values.

```
lat_long = {
    "HW": (41.88609733324964, -87.62682604591349),
    "HT": (41.9646769664519, -87.65901943241516),
    "KK": (41.77847328856264, -87.6435785385309),
    "MX": (41.87800548491064, -87.67453475017268),
    "OH": (41.71006715754713, -87.5886722734757),
    "RD": (41.75677704810169, -87.72315805813204),
    "WW": (41.95836512405638, -87.78738482318016)
}
lat_long
```

Since the Earth is curved, we cannot use the Euclidean distance. We will use the Haversine formula instead. Note that the Haversine formula still produces an approximation because the Earth is not a perfect sphere. Here is a function to compute the Haversine formula.

```
def haversine(lat1, lon1, lat2, lon2):
    '''Reference:␣
        https://cs.nyu.edu/~visual/home/proj/tiger/gisfaq.html'''
    import math

    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1,
        lat2, lon2])

    dlat = lat2 - lat1
    dlon = lon2 - lon1

    a = math.sin(dlat / 2)**2 + \
        math.cos(lat1) * math.cos(lat2) * math.sin(dlon /
            2)**2

    c = 2 * math.asin(min(1.0, math.sqrt(a)))

    # Earth's approximate radius in kilometers
    R = 6367.0

    distance = R * c

    return distance

print("Ready␣to␣use␣`haversine()`")
```

Now we can make an edge `list`. We will represent each campus as a node with the initials of the college name. The weight of the edge will represent the Haversine value between the locations. For example, express the route between Harold Washington College and Harry Truman College as `("HW", "HT", Haversine)`.

```
distances = []
colleges = list(lat_long.items())
for i in range(len(colleges)):
    college1, (lat1, lon1) = colleges[i]
    for j in range(i + 1, len(colleges)):
        college2, (lat2, lon2) = colleges[j]
        dist = haversine(lat1, lon1, lat2, lon2)
        distances.append((college1, college2, dist))

print("\nDistances␣between␣colleges␣(in␣kilometers):")
for edge in distances:
    college1, college2, dist = edge
    print(f"{college1}␣-␣{college2}:␣{dist:.2f}␣km")
```

8.3.3 Graph

Swap $(Latitude, Longitude)$ coordinates for plotting with (x, y) coordinates.

```
pos = {college: (lon, lat) for college, (lat, lon) in
    lat_long.items()}
```

```
pos
```

Create a `Graph` from the edge `list`:

```
G = Graph(distances)
G.show(
    pos=pos,  # Positions are (longitude, latitude)
    edge_labels=True,
    vertex_size=500,
    figsize=20,
    title="CCC␣Distance␣Graph"
)
```

8.3.4 Railway

So far, we have encountered various concepts for connecting a graph's vertices, including the Hamilton path and the MST. Let's consider what technique is best suited for solving the problem of constructing a railway that optimizes material costs.

The previous chapter used the traveling salesperson algorithm to optimize a delivery route. Since we aim to optimize material costs, you might think of following a similar approach: apply the traveling salesperson algorithm, eliminate the greatest edge from the Hamilton circuit, and design the railway with the minimum Hamilton path. If we take a Hamilton circuit and eliminate one edge, we obtain a spanning tree. While the Hamilton path optimizes graph traversal by visiting each vertex exactly once in a single path, it does not guarantee that all vertices are connected with the minimal total weight.

In a Hamilton path, the requirement to visit each vertex in a single path can force the inclusion of high-weight edges. Alternatively, the MST is not restricted by the requirement of connecting vertices with a path. The MST can avoid high-weight edges by connecting vertices without regard to forming a path as long as the graph remains connected and acyclic. Although the minimum Hamilton path is one of many possible spanning trees, it is not an MST. Prim's Algorithm ensures the weight of the spanning tree is minimal because, at each iteration, it selects the smallest-weight edge.

Let's find the MST edge `list` of the campus locations with the `min_spanning_tree(by_weight=True)` function.

```
mst = G.min_spanning_tree(by_weight=True)
mst
```

Visualize the MST with the vertex positions mapped to the geographical coordinates of each campus location.

```
Graph(mst).show(
    pos=pos,
    edge_labels=True,
    vertex_size=500,
    figsize=15,
    title="CCC␣Minimum␣Spanning␣Tree"
)
```

8.3.5 Conclusion

In this exercise, we only optimized construction material costs. In a real-world

scenario, we may want to create a railway that optimizes both travel time and material costs. In the case of the Chicago L train system, the railway resembles a tree when ignoring the downtown Loop. The L receives criticism for its lack of interconnectivity. For example, finding an efficient route connecting the end of the Blue Line with the end of the Red Line is challenging because a traveler may need to commute all the way downtown from one end of the railway to reach another end. As an interesting challenge, you can optimize both travel time and construction costs.

Chapter 9

Lattices

This chapter builds on the partial order sets introduced earlier and explains how to ask Sage to decide whether a given poset is a lattice. Then, we show how to calculate the meet and join tables using built-in and customized Sage functions.

9.1 Lattices

9.1.1 Definition

A **lattice** is a partially ordered set (**poset**) in which any two elements have a least upper bound (also known as join) and greatest lower bound (also known as meet).

In Sage, a lattice can be represented as a poset using the `Poset()` function. This function takes a tuple as its argument, where the first element is the set of elements in the poset, and the second element is a list of ordered pairs representing the partial order relations between those elements.

First, let's define the lists of elements and relations we will use for the following examples:

```
elements = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

relations = [
    ['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
    ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']
]
print("Elements: ", elements)
print("Relations: ", relations)
```

Create a poset from a tuple of elements and relations.

```
PO = Poset((elements, relations))
PO.show()
```

The function `is_lattice()` determines whether the poset is a lattice.

```
PO.is_lattice()
```

Notes. Lattices have practical applications in computer science, such as static program analysis and distributed programming.

We can also use `LatticePoset()` function to plot the lattice. The function `Poset()` can be used with any poset, even when the poset is not a lattice. The `LatticePoset()` function will raise an error if the poset is not a lattice.

```
LP = LatticePoset((elements, relations))
LP.show()
```

9.1.2 Join

The join of two elements in a lattice is the least upper bound of those elements.

To check if a poset is a join semi-lattice (every pair of elements has a least upper bound), we use `is_join_semilattice()` function.

```
PO.is_join_semilattice()
```

We can also find the join for individual pairs using the `join()` function.

```
PO.join('b', 'f')
```

9.1.3 Meet

The meet of two elements in a lattice is their greatest lower bound.

To check if a poset is a meet semi-lattice (every pair of elements has a greatest lower bound), we use `is_meet_semilattice()` function.

```
PO.is_meet_semilattice()
```

We can also find the meet for individual pairs using the `meet()` function.

```
PO.meet('a', 'b')
```

9.1.4 Divisor Lattice

The Sage `DivisorLattice()` function returns the divisor lattice of an integer.

The elements of the lattice are divisors of n and $x < y$ in the lattice if x divides y.

```
Posets.DivisorLattice(12).show()
```

9.2 Tables of Operations

This section examines the representation of meet ( ) and join ( ) operations within lattices using operation tables.

9.2.1 Meet Operation Table

The meet operation table illustrates the greatest lower bound, or meet, for every pair of elements in the lattice.

To output the table as a matrix, we need to specify that the poset is indeed a lattice, thus requiring us to use the function `LatticePoset()`. Then, we can use the function `meet_matrix()` to process the table.

```
elements = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

relations = [
    ['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'd'],
    ['c', 'e'], ['d', 'f'], ['e', 'f'], ['f', 'g']
]

L = LatticePoset((elements, relations))
M = L.meet_matrix()
show(M)
```

From the output matrix, we can see that each entry a_{ij} is not the actual value of the meet of the elements a_i and a_j but just its position in the lattice. Let's show the values:

```
linear_extension = L.linear_extension()

values_meet_matrix = [
    [
        linear_extension[M[i, j]]
        for j in range(len(elements))
    ]
    for i in range(len(elements))
]

values_meet_matrix
```

Show the output as a table:

```
import pandas as pd

df = pd.DataFrame(
    values_meet_matrix,
    index=elements,
    columns=elements
)

df
```

9.2.2 Join Operation Table

Conversely, the join operation table presents the least upper bound, or join, for each pair of lattice elements.

```
J = L.join_matrix()

show(J)
```

Output the elements of the poset:

```
linear_extension = L.linear_extension()

values_join_matrix = [
    [
        linear_extension[J[i, j]]
        for j in range(len(elements))
    ]
```

```
    for i in range(len(elements))
]

values_join_matrix
```

Show the output as a table instead of a matrix.

```
import pandas as pd

df = pd.DataFrame(
    values_join_matrix,
    index=elements,
    columns=elements
)

df
```

Chapter 10

Boolean Algebra

This chapter completes the preceding one by explaining how to ask Sage to decide whether a given lattice is a Boolean algebra. We also illustrate basic operations with Boolean functions.

10.1 Boolean Algebra

A Boolean algebra is a bounded lattice that is both complemented and distributive. Let's define the `is_boolean_algebra()` function to determine whether a given poset is a Boolean algebra. The function accepts a finite partially ordered set as input and returns a tuple containing a boolean value and a message explaining the result. Run the following cell to define the function and call it in other cells.

```
def is_boolean_algebra(P):
    try:
        L = LatticePoset(P)
    except ValueError as e:
        return False, str(e)
    if not L.is_bounded():
        return False, "The lattice is not bounded."
    if not L.is_distributive():
        return False, "The lattice is not distributive."
    if not L.is_complemented():
        return False, "The lattice is not complemented."
    return True, "The poset is a Boolean algebra."
```

Let's check if the following poset is a Boolean algebra.

```
S = Set([1, 2, 3, 4, 5, 6])

P = Poset((S, attrcall("divides")))

show(P)
```

```
is_boolean_algebra(P)
```

When we pass `P` to the `is_boolean_algebra()` function, `LatticePoset()` raises an error because `P` is not a lattice. The `ValueError` provides more

information about the absence of a top element. Therefore, `P` is not a Boolean algebra.

```
T = Subsets(['a', 'b', 'c'])

Q = Poset((T, lambda x, y: x.issubset(y)))

Q.plot(vertex_size=3500, border=True)
```

```
is_boolean_algebra(Q)
```

Let's examine the divisor lattice of 30:

```
dl30 = Posets.DivisorLattice(30)
show(dl30)
is_boolean_algebra(dl30)
```

Now for the divisor lattice of 20:

```
dl20 = Posets.DivisorLattice(20)
show(dl20)
is_boolean_algebra(dl20)
```

Here is a classic example in the field of computer science:

```
B = posets.BooleanLattice(1)
show(B)
is_boolean_algebra(B)
```

10.2 Boolean functions

A **Boolean function** is a function that takes only values 0 or 1 and whose domain is the Cartesian product $\{0, 1\}^n$.

Notes. Boolean algebra influences the design of digital circuits. For example, simplifying a digital circuit can minimize the number of gates used and reduce the manufacturing cost.

A **minterm** of the Boolean variables $x_1, x_2, \ldots, x_n$ is the Boolean product $y_1 \cdot y_2 \cdot \ldots \cdot y_n$ where each $y_i = x_i$ or $y_i = \overline{x_i}$.

A sum of minterms is called a **sum-of-products** expansion. In this section, we will examine various methods for finding the sum-of-products expansion of a Boolean function.

To find the sum-of-products expansion using a truth table, we first convert the `truthtable()` into a form that is iterable with `get_table_list()`. For every row where the output value is `True`, we construct a minterm:

- Include the variable as is if its value is `True`
- Include the negation of the variable if its value is `False`
- The `zip` function pairs each variable with its corresponding value, allowing us to create minterms efficiently.
- We add each minterm to the `sop_expansion list` using the `&` operator.
- Finally, we join all minterms with the `|` operator to form the sum-of-products expansion.

- The function returns the sum-of-products expansion as a `sage.logic.boolformula.BooleanFormula` instance.

```
def truth_table_sop(expression):
    # Check if the input is a string, and if so, convert it
        to a formula object
    if isinstance(expression, str):
        h = propcalc.formula(expression)
    elif isinstance(expression,
        sage.logic.boolformula.BooleanFormula):
        h = expression
    else:
        raise ValueError

    table_list = h.truthtable().get_table_list()
    sop_expansion = []

    for row in table_list[1:]:  # Skip the header row
        if row[-1]:  # If the output value is True
            minterm = []
            for var, value in zip(table_list[0], row[:-1]):
                # Iterate over each variable and its value
                in this row
                if value:
                    minterm.append(var)  # Include variable
                        as is if True
                else:
                    minterm.append(f'~{var}')  # Include the
                        negated variable if False
            sop_expansion.append('␣&␣'.join(minterm))  #
                Join variables in the minterm using the AND
                operator

    sop_result = '␣|␣'.join(f'({m})' for m in sop_expansion)
         # Join minterms using the OR operator
    return propcalc.formula(sop_result)
```

For your convenience, our `truth_table_sop` function converts `String` input with `propcalc.formula`. Therefore, the input accepts `String` representations of Boolean expressions. Alternatively, you may pass an instance of `sage.logic.boolformula.BooleanFormula` directly to the function.

```
truth_table_sop("x␣&␣(y␣|␣z)")
```

Let's verify that the sum-of-products expansion we found with the truth table is equivalent to the original expression.

```
truth_table_sop("x␣&␣(y␣|␣z)") == propcalc.formula("x␣&␣(y␣|␣
   z)")
```

Our `sop_expansion` function mimics the manual process of finding the sum-of-products expansion of a Boolean function. This process does not guarantee the minimal form of the Boolean expression.

If we dig around in the Sage source code, we can find a commented-out `Simplify()` function that relied on the `Boolopt` package and the Quine-McCluskey algorithm. The Quine-McCluskey algorithm guarantees the minimal form of the Boolean expression, but the exponential complexity of the algorithm makes it impractical for large expressions. Moreover, in the Sage documentation, we

see a placeholder function called `Simplify()` that returns a `NotImplementedError` message. The Sage community is waiting for someone to implement this function with the Espresso algorithm. While the Espresso algorithm does not guarantee the minimal form of the Boolean expression, it is more efficient than the Quine-McCluskey algorithm.

Sage integrates well with Python libraries like SymPy, which have built-in functions for Boolean simplification. The SymPy `SOPform` function takes the variables as the first argument and the minterms as the second argument. The function returns the sum-of-products expansion of the Boolean function in the smallest sum-of-products form. To use the SymPy `SOPform` function in Sage, first extract the variables and minterms of an expression.

We extract the variables from the first row of the truth table.

```
expression = propcalc.formula("x␣&␣(y␣|␣z)")
table_list = expression.truthtable().get_table_list()
variables = table_list[0]
print(variables)
```

We make the variables compatible with the SymPy `SOPform` function by converting them to SymPy symbols.

```
from sympy import symbols
sympy_variables = symbols('␣'.join(variables))
print(sympy_variables)
```

We extract the minterms from the rows where the output is `True`.

```
minterms = [row[:-1] for row in table_list[1:] if row[-1]]
print(minterms)
```

Now that we have the variables and minterms, we can use the SymPy `SOPform` function to find the sum-of-products expansion of the Boolean function.

```
from sympy.logic import SOPform
from sympy import symbols

def sympy_sop(expression):
    # Convert input expression to Sage formula object if
        necessary
    if isinstance(expression, str):
        formula_object = propcalc.formula(expression)
    elif isinstance(expression,
        sage.logic.boolformula.BooleanFormula):
        formula_object = expression
    else:
        raise ValueError("Invalid␣input:␣expression␣must␣be␣
            a␣string␣or␣a␣BooleanFormula␣object.")

    # Generate the truth table from the formula object
    truth_table = formula_object.truthtable()
    table_list = truth_table.get_table_list()

    # Extract variables and minterms from the truth table
    variables = table_list[0]
    minterms = [row[:-1] for row in table_list[1:] if
        row[-1]]
```

```
    # Convert Sage variables to SymPy symbols
    sympy_variables = symbols('␣'.join(variables))

    # Use SymPy to compute the SOP form
    sop_result = SOPform(sympy_variables, minterms)

    return propcalc.formula(str(sop_result))
```

```
sympy_sop("x␣&␣(y␣|␣z)")
```

Let's verify that the sum-of-products expansion we found with SymPy is equivalent to the original expression.

```
sympy_sop("x␣&␣(y␣|␣z)") == propcalc.formula("x␣&␣(y␣|␣z)")
```

Now, we present a manual method for finding the sum of products by applying the Boolean identities. Let's find the sum-of-products expansion of the Boolean function

$$h(x, y) = x + \bar{y}.$$

We can apply the Boolean identities and use Sage to verify our work. Currently, we have a sum of two terms but no products. We can apply the identity law to introduce the product terms. Now, we have the equivalent expression

$$h(x, y) = x \cdot 1 + 1 \cdot \bar{y}.$$

Warning: Do not attempt to apply the identity law or null law within the `formula` function. If you try to directly apply the identity law within the formula function like so, `propcalc.formula("x & 1 | 1 ~y")`, Sage will raise an error because `propcalc.formula` interprets 1 as a variable. Variables cannot start with a number.

The `formula` function only supports variables and the following operators:

- `&` *and*
- `|` *or*
- `~` *not*
- `^` *xor*
- `->` *if then*
- `<->` *if and only if*

```
h = propcalc.formula("x␣|␣~y")
show(h)
```

Apply the complement law and verify that our transformed expression is equivalent to the original expression.

```
h_complement = propcalc.formula("x␣&␣(y␣|␣~y)␣|␣(x␣|␣~x)␣&␣␣
    ~y")
show(h_complement)
h_complement == h
```

Apply the distributive law and verify that our transformed expression is equivalent to the original expression.

```
h_distributive = propcalc.formula("x␣&␣y␣|␣x␣&␣~y␣|␣x␣&␣~y␣|␣
    ~x␣&␣~y")
show(h_distributive)
h_distributive == h
```

Apply the idempotent law and verify that our transformed expression is equivalent to the original expression.

```
h_idempotent = propcalc.formula("x␣&␣y␣|␣x␣&␣~y␣|␣~x␣&␣~y")
show(h_idempotent)
h_idempotent == h
```

We started with the expression,

$$h(x, y) = x + \bar{y}$$

After applying the identity, complement, and distributive laws, we transformed the Boolean function into the sum-of-products expansion

$$h(x, y) = x \cdot y + x \cdot \bar{y} + x \cdot \bar{y} + \bar{x} \cdot \bar{y}.$$

Chapter 11

Logic Gates

This chapter explains how to process binary inputs in Sage to produce specific outputs based on basic logic gates, such as AND, OR, and NOT. Then, we show how these gates combine to form more complex circuits and integrate into everyday electronics, using built-in and customized Sage functions to simulate and analyze their behavior.

11.1 Logic Gates

Logic gates are the foundation of digital circuits. They process binary inputs to produce specific outputs. The basic logic gates are AND, OR, and NOT. Derived gates include $NAND$, NOR, XOR, and $XNOR$. Each gate has its own symbol and behavior defined by a truth table.

Notes. Logic gates combine to form complex systems such as CPUs and memory circuits.

11.1.1 AND Gate

The `AND` gate produces a 1 only when both inputs are 1.

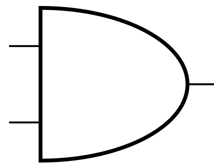

Figure 11.1.1 AND Gate

```
from sympy.logic.boolalg import And
from sympy.abc import A, B
And(A, B)
```

Truth table for the `AND` gate:

```
# Generate truth table for AND gate
print("\nA␣|␣B␣|␣A␣AND␣B")
print("--|---|--------")
for A in [False, True]:
    for B in [False, True]:
        print(f"{int(A)}␣|␣{int(B)}␣|␣{int(bool(And(A,␣
```

```
            B)))}")
```

11.1.2 OR GATE

The `OR` gate produces a `1` if at least one input is `1`.

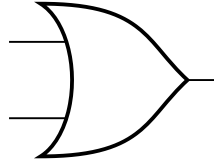

Figure 11.1.2 OR Gate

```
from sympy.logic.boolalg import Or
from sympy.abc import A, B
Or(A, B)
```

Truth table for the `OR` gate:

```
# Generate truth table for OR gate
print("\nA | B | A OR B")
print("--|---|--------")
for A in [False, True]:
    for B in [False, True]:
        print(f"{int(A)} | {int(B)} | {int(bool(Or(A, B)))}")
```

11.1.3 NOT Gate

The `NOT` gate inverts the input: `1` becomes `0`, and `0` becomes `1`.

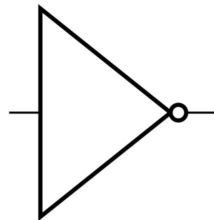

Figure 11.1.3 NOT Gate

```
from sympy.logic.boolalg import Not
from sympy.abc import A
Not(A)
```

Truth table for the `NOT` gate:

```
# Generate truth table for NOT gate
print("\nA | NOT A")
print("--|-------")
for A in [False, True]:
    print(f"{int(A)} | {int(bool(Not(A)))}")
```

11.1.4 NAND Gate

`NAND`: Produces `0` only when both inputs are `1`.

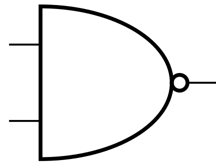

Figure 11.1.4 NAND Gate

11.1.5 NOR Gate

`NOR`: Produces 1 only when both inputs are `0`.

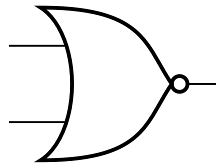

Figure 11.1.5 NOR Gate

11.1.6 XOR Gate

`XOR`: Produces 1 when inputs differ.

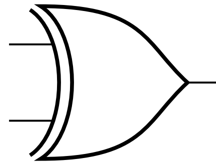

Figure 11.1.6 XOR Gate

11.1.7 XNOR Gate

`XNOR`: Produces 1 when inputs are the same.

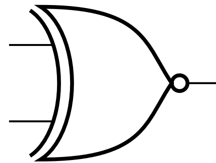

Figure 11.1.7 XNOR Gate

```
from sympy.logic.boolalg import And, Or, Not, Xor

def nand(A, B):
    return Not(And(A, B))

def nor(A, B):
    return Not(Or(A, B))

def xor(A, B):
    return Xor(A, B)

def xnor(A, B):
    return Not(Xor(A, B))

# User-defined inputs
A = 1 # Replace with 0 or 1 for input A
```

```
B = 0 # Replace with 0 or 1 for input B
gate = "xor" # Replace with "nand", "nor", "xor", or "xnor"


if gate == "nand":
    result = nand(A, B)
elif gate == "nor":
    result = nor(A, B)
elif gate == "xor":
    result = xor(A, B)
elif gate == "xnor":
    result = xnor(A, B)
else:
    result = "Invalid␣gate␣type!␣Please␣use␣'nand',␣'nor',␣
        'xor',␣or␣'xnor'."

result
```

11.2 Combinations of Logic Gates

Logic gates can be combined to create more complex circuits that perform specific tasks. By linking gates together, we can create circuits that process multiple inputs to produce a desired output. For example, combining an `AND` gate and a `NOT` gate results in a `NAND` gate, which inverts the output of the `AND` gate. More complex circuits, such as half-adders and multiplexers, are built by combining basic gates in strategic ways.

Let's look at a circuit. We evaluate this circuit by setting True for X, Y, and False for Z below using Sage.

```
from sympy.logic.boolalg import And, Or, Not
from sympy.abc import X, Y, Z

# Define the logic circuit
F = Or(And(Not(X), Y, Z), And(X, Not(Y), Z), And(X, Y,
    Not(Z)), And(X, Y, Z))

# Evaluate the logic circuit with values for X, Y, and Z
circuit_output = F.subs({X: True, Y: True, Z: False})
circuit_output
```

Boolean algebra provides a way to simplify complex logic circuits. By using Boolean algebra rules, you can take a complicated circuit and reduce it to a simpler form without changing its functionality.

Here's a practical example. Consider the following Boolean expression, which combines several gates:

```
# Original Boolean expression
from sympy import simplify
from sympy.logic.boolalg import And, Or, Not
from sympy.abc import x, y, z

# Define the expression
D = Or(And(Not(x), y, z), And(x, Not(y), z), And(x, y,
    Not(z)), And(x, y, z))
D
```

```
# Simplified Boolean expression
simplified_D = simplify(D)
simplified_D
```

Truth tables are a visual way to represent how inputs to a logic circuit map to its outputs. For each possible combination of inputs, the table shows the corresponding outputs, making it easier to analyze and understand the behavior of the circuit.

Let's create a truth table for the simplified circuit.

$F = (x\ and\ y)\ or\ (x\ and\ z)\ or\ (y\ and\ z)$

Here, we will show the intermediary steps to find the final output of the function.

```
from sympy.logic.boolalg import And, Or, truth_table
from sympy.abc import x, y, z

# Define the logic function
intermediate1 = And(x, y) # x AND y
intermediate2 = And(x, z) # x AND z
intermediate3 = And(y, z) # y AND z

# (x AND y) OR (x AND z) OR (y AND z)
final_output = Or(intermediate1, intermediate2,
    intermediate3)

# Variables and expressions
variables = [x, y, z]
expressions = [intermediate1, intermediate2, intermediate3,
    final_output]

# Header names and column widths
headers = ["x", "y", "z", "x AND y", "x AND z", "y AND z",
    "F"]
column_widths = [5, 5, 5, 10, 10, 10, 5] # Adjust widths as
    needed

# Print header row with adjusted spacing
header_row = " | ".join(h.ljust(w) for h, w in zip(headers,
    column_widths))
print(header_row)
print("-" * len(header_row))

# Generate and print the truth table rows
for row in truth_table(final_output, variables):
    inputs = row[0]
    outputs = [int(bool(expr.subs(dict(zip(variables,
        inputs))))) for expr in expressions]
    table_row = " | ".join(str(int(bool(x))).ljust(w) for x,
        w in zip(list(inputs) + outputs, column_widths))
    print(table_row)
```

Chapter 12

Finite State Machines

This chapter delves into a powerful abstract model, namely the *finite-state machines*. Beyond the theoretical framework, the content of this chapter demonstrates the use of Sage to define, model, build, visualize, and execute examples of state machines, showcasing their application in solving real-world problems.

12.1 Definitions and Components

The defining feature of any abstract machine is its memory structure, ranging from a *finite* set of states in the case of finite-state machines to more complex memory systems (e.g., *Turing machines* and *Petri nets*).

A **Finite-State Machine (FSM)** is a computational model that has a finite set of possible states S, a finite set of possible input symbols (the input alphabet) X, and a finite set of possible output symbols (the output alphabet) Z. The machine can exist in one of the states at any time, and based on the machine's input and its current state, it can transition to any other state and produce an output. The functions that take in the machine's current state and its input and map them to the machine's future state and its output are referred to as the *state transition* function and the *output* function, respectively. The default state of an FSM is referred to as the *initial state*.

12.1.1 Mealy State Machine

A Mealy finite-state machine is defined by the tuple (S, X, Z, w, t, s_0) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, a finite set that corresponds to the set of all memory configurations that the machine can have at any time.
- The state s_0 is called the *initial state*.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $Z = \{z_0, z_1, z_2, \ldots, z_k\}$ is the output alphabet.
- $w : S \times X \to Z$ is the output function, which specifies which output symbol $w(s, x) \in Z$ is written onto the output device when the machine is in state s and the input symbol x is read.
- $t : S \times X \to S$ is the next-state (or transition) function, which specifies which state $t(s, x) \in S$ the machine should move to when it is currently in state s and it reads the input symbol x.

12.1.2 Other Types of Finite State Machines

12.1.2.1 Moore Machine

In a **Moore Machine**, the output depends *solely* on the current state. Unlike Mealy state machine, this machine must enter a new state for the output to change.

A Moore machine is also represented by the 6-tuple (S, X, Z, w, t, s_0) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, and s_0 is the initial state.
- The state s_0 is called the *initial state*.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $Z = \{z_0, z_1, z_2, \ldots, z_k\}$ is the output alphabet.
- $w : S \to Z$ is the output function, which specifies which output symbol $w(s) \in Z$ associated with the machine current state s.
- $t : S \times X \to S$ is the transition function, which specifies which next state $t(s, x) \in S$ the machine should move to when its current state is s and it has the input symbol x.

12.1.2.2 Finite-State Automaton

A *final state* (also known as the accepted state) is defined as a *special* predefined state that indicates whether an input sequence is valid or accepted by the finite-state machine. The set F of all final states is a subset of the states set S.

A **Finite-State Automaton** is a finite-state machine *with no output*, and it is represented by the 5-tuple (S, X, t, s_0, F) where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is the state set, s_0 is the initial state, and F is the set of final states.
- The state s_0 is called the *initial state*.
- The subset $F \subset S$ is the set of all final states of the machine.
- $X = \{x_0, x_1, x_2, \ldots, x_m\}$ is the input alphabet.
- $t : S \times X \to S$ is the transition function, which specifies which next state $t(s, x) \in S$ the machine should move to when its current state is s and it has the input symbol x.

When the state machine processes a finite input sequence, it transitions through various states based on each input in the sequence and the current state of the machine. If, after processing the entire sequence, the machine lands in any of the *final states*, then the input sequence is considered valid (or recognized according to the machine's rules). Otherwise, the input sequence is rejected as invalid.

12.1.2.3 Deterministic Finite Automaton (DFA)

A **Deterministic Finite Automaton (DFA)** is a simplified automaton in which each state has exactly one transition for each input. DFAs are typically used for lexical analysis, language recognition, and pattern matching.

Note. A text parser or a string-matching application that recognizes a specific language or regular expressions are real-world examples of DFA use.

12.1.2.4 Nondeterministic Finite Automaton (NFA)

Unlike a DFA, an **NFA** allows multiple transitions for the same input or even transitions without consuming input (ϵ-transitions).

12.1.2.5 Turing Machine

A **Turing Machine** is an expansion of an FSM, which includes infinite tape memory representing both the input and output streams (shared stream). Unlike all other FSMs, a Turing machine can alter the input/output stream, and as such, it is capable of simulating any algorithm. Turing machines are the theoretical foundation for modern computation (any general-purpose computer executing any algorithm can be modeled as a Turing Machine).

Finite state machines are a foundational concept in computer science, often associated with tasks related to system designs (circuits and digital computers, algorithms, etc.). However, the vast and rich domain of applications of state machines extends far beyond simple simulations to the full control logic of complex industrial processes and workflows. These tasks can vary in complexity, ranging from a simple parity check to managing traffic patterns, a programming language compiler, or natural language recognition and processing.

State machines offer a structured way to model systems with discrete states and transitions. Different variants, such as the Mealy machine and Moore machine, have distinct characteristics and, as such, can adapt to various applications.

12.2 Finite State Machines in Sage

Although Sage includes a dedicated built-in rich module to handle various types of state machines, it may not always be sufficient to address certain use cases or implement specific custom behaviors of the machine. Additionally, the built-in module allows state machines to be defined and constructed in different ways, providing greater flexibility and making it more suitable from a programmer's perspective. However, it may not fully conform to the precise definition given earlier. This highlights that it is still possible to model, construct, display, and run relatively simple state machines by utilizing general-purpose tools, such as graphs and transition matrices, to represent and operate on state machines.

Notes. While Sage provides basic tools to represent and simulate state machines, it may not natively support more complex state machine features such as parallel states or hierarchical transitions.

12.2.1 The Elevator State Machine

Let's design a basic controller to an elevator to show the process of defining states, creating a state transition graph, visualizing the state machine, and simulating its execution in Sage.

Consider a 3-level elevator (floors 1 through 3). The elevator has 3 buttons for users to select the destination floor (only one can be selected at a time). Depending on the current position and the selected floor, the elevator can go up, go down, or remain on the same floor.

12.2.2 Description of the Elevator FSM

This elevator system can be modeled and simulated using a finite-state machine with states $S = \{f_1, f_2, f_3\}$ representing each floor, the user inputs set $X = \{b_1, b_2, b_3\}$ (where b_i represents the button for i^{th} floor), and the outputs set $Z = \{U, D, N\}$ for 'going up', 'going down', or 'going nowhere'.

The components of this FSM are transcribed in the following table.

Table 12.2.1 The Elevator State Machine Definition

	next			output		
current	b_1	b_2	b_3	b_1	b_2	b_3
f_1	f_1	f_2	f_3	N	U	U
f_2	f_1	f_2	f_3	D	N	U
f_3	f_1	f_2	f_3	D	D	N

The following steps outline the approach to build and test the elevator controller system:

1. Define the elements of the Finite State Machine: States, Inputs, Transitions, and Outputs.
2. Construct the State Machine.
3. Run the machine using a sample input set.

12.2.3 Elements of the Elevator FSM

The first step is to define the states and transitions in the state machine, which can be represented using lists and dictionaries.

```
# Define state, input and output sets
states = ['f1', 'f2', 'f3']
inputs = ['b1', 'b2', 'b3']
outputs = ['U', 'D', 'N']

# Transitions as a dictionary {(current_state, input):
    next_state}
transitions = {
    ('f1', 'b1'): 'f1',
    ('f1', 'b2'): 'f2',
    ('f1', 'b3'): 'f3',

    ('f2', 'b1'): 'f1',
    ('f2', 'b2'): 'f2',
    ('f2', 'b3'): 'f3',

    ('f3', 'b1'): 'f1',
    ('f3', 'b2'): 'f2',
    ('f3', 'b3'): 'f3',
}

# The machine outputs control how the elevator would move
outputs = {
    ('f1', 'b1'): 'N',
    ('f1', 'b2'): 'U',
    ('f1', 'b3'): 'U',

    ('f2', 'b1'): 'D',
    ('f2', 'b2'): 'N',
```

```
    ('f2', 'b3'): 'U',

    ('f3', 'b1'): 'D',
    ('f3', 'b2'): 'D',
    ('f3', 'b3'): 'N',
}

# Display the machine configuration
print('States:␣', states)
print('Transitions:␣', transitions)
print('Outputs:␣', outputs)
```

12.2.4 Graph Model of the Elevator FSM

An FSM can be modeled as a graph where vertices represent the states, and the directed edge between vertices is the relationship between two states (the transition from one state to the other). The weight of a directed edge between two vertices represents the pair of input and output associated with the transition between the two states.

In Sage, the `DiGraph` class can be used to represent the states, transitions, and outputs of the state machine as a directed graph, leveraging the graph structure to visualize the state machine representation.

```
# 'DiGraph' is imported by default. If not, it can be
    imported as follow
# from sage.graphs.digraph import DiGraph

# Initialize a directed graph
elevator_fsm = DiGraph(loops=True)

# Add states as vertices
elevator_fsm.add_vertices(states)

# Add transitions and outputs as edges
for (_state, _input), next_state in transitions.items():
    _output = outputs[(_state, _input)]
    edge_label = f"{_input},␣{_output}"
    elevator_fsm.add_edge(_state, next_state,
        label=edge_label)

# Display the graph (state machine)
elevator_fsm.show(
    figsize=[5.6, 5.6],
    layout='circular',
    vertex_size=250,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0,
)
```

The `show()` method renders a graphical representation of the state machine. Each vertex in the graph represents a state, and each directed edge represents a transition, labeled as (input, output).

12.2.5 Run the Elevator State Machine

Next, the state machine's behavior can be simulated by defining a function that processes a list of inputs and transitions through the states accordingly.

```
# Function to run the state machine
def run_state_machine(start_state, inputs):
    current_state = start_state
    for _input in inputs:
        print(f"Current␣state:␣{current_state},␣Input:␣
            {_input}")

        if (current_state, _input) in transitions:
            current_output = outputs[(current_state, _input)]
            current_state = transitions[(current_state,
                _input)]
            print(
                f"Transitioned␣to:␣{current_state}\n"
                f"Output:␣{current_output}\n"
            )
        else:
            print(
                f"No␣transition/output␣available␣for␣input␣
                    {_input}␣in␣state␣{current_state}"
            )
            break

    print(f"Last␣state:␣{current_state}")

# Example of running the state machine
start_state = 'f2'
inputs = ['b1', 'b1', 'b3', 'b2']

run_state_machine(start_state, inputs)
```

The `run_state_machine()` function simulates the state machine by processing a list of inputs starting from an initial state.

12.2.6 The Traffic Light State Machine

Let's design a simple traffic light controller to illustrate alternative methods for defining, visualizing, and executing finite state machines in Sage.

Consider a simplified traffic light system controlled by preset timers. This system operates through three phases that represent the flow of road traffic: Free-flowing, Slowing-down, and Halted. These phases correspond to the traffic light signals: green, yellow, and red, controlling the flow of traffic. The system uses three timer settings: 30 seconds, 20 seconds, and 5 seconds. When a timer expires, it triggers the transition to the next phase. Initially, the light is green, the traffic is flowing, and:

- When the 30-seconds timer expires, the traffic light changes from green to yellow, and traffic begins to slow down.
- When the 5-seconds timer expires, the traffic light changes from yellow to red, bringing traffic to a complete stop.
- When the 20-seconds timer expires, the traffic light changes from red to green, allowing traffic to start moving again.

12.2.7 Description of the Traffic Light FSM

In this traffic light system, the three phases representing the flow of road traffic: *Free-flowing (F)*, *Slowing-down (S)*, and *Halted (H)* are the states $S = \{F, S, H\}$ of the FSM. These phases correspond to the traffic light signals: green (G), yellow (Y), and red (R), which are the outputs set $Z = \{G, Y, R\}$ of the system. The timers driving the transitions are the inputs set $X = \{t_{5s}, t_{20s}, t_{30s}\}$ of this traffic light system.

The following table summarize the elements of the traffic light FSM.

Table 12.2.2 The Traffic Light State Machine Definition

	next			**output**		
current	t_{5s}	t_{20s}	t_{30s}	t_{5s}	t_{20s}	t_{30s}
F	F	F	S	G	G	Y
S	H	S	S	R	Y	Y
H	H	F	H	R	G	R

By applying the same steps and approach as in the previous section, the traffic light controller system will be built and tested, this time utilizing the Sage built-in module and functions.

12.2.8 Using `FiniteStateMachine' Module

Sage `FiniteStateMachine` built-in library provides a powerful tool to model, construct as well as simulate state machines of various systems. This module will be leveraged to showcase its capabilities on the given example, and demonstrating how it can be used to construct and display the FSM, manage its state transitions and outputs.

The command `FiniteStateMachine()` constructs an *empty* state machine (no states, no transitions).

```
from sage.combinat.finite_state_machine import FSMState

# FSM states, inputs and outputs
states = ['F', 'S', 'H']         # Free-flowing,
    Slowing-down, Halted
inputs = ['t30s', 't5s', 't20s'] # timer durations before
    state transitions
outputs = ['G', 'Y', 'R']        # traffic light: Green,
    Yellow, Red

# Create an empty state machine object
traffic_light_fsm = FiniteStateMachine()
traffic_light_fsm
```

The function `FSMState()` defines a state for a given label. The `is_initial` flag can be set to true to set the current state as the *initial state* of the finite state machine. The method `add_state()` appends the created state to an existing state machine.

```
# Define a new state then adding it
free_flowing = FSMState('F', is_initial=True)
traffic_light_fsm.add_state(free_flowing)

# Adding more states by their labels (saving state handlers,
    to use them in state transitions)
```

```
slowing_down = traffic_light_fsm.add_state('S')
halted = traffic_light_fsm.add_state('H')

# the FiniteStateMachine instance
traffic_light_fsm
```

To check whether or not a finite state machine has a state defined, `has_state()` method can be used by passing in the state label (case-sensitive).

```
traffic_light_fsm.has_state('F')
```

The function `states()` enumerates the list of all defined states of the state machine.

```
traffic_light_fsm.states()
```

The method `initial_states()` lists the defined initial state(s) of the state machine.

```
traffic_light_fsm.initial_states()
```

To define a new transition between two states (as well as the input triggering the transition, and the output associated with the state transition), the method `FSMTransition()` can be used. The method `add_transition()` attaches the defined transition to the state machine, and the function `transitions()` enumerates the list of all defined transitions of the state machine.

```
from sage.combinat.finite_state_machine import FSMTransition

# defining 3 transitions, and associating them the state
    machine
# After 30sec, transition from free-flowing to slowing-down,
    and set traffic light to yellow
traffic_light_fsm.add_transition(
    FSMTransition(
        from_state=free_flowing,
        to_state=slowing_down,
        word_in='t30s',
        word_out='Y'
    )
)

# After 5sec, transition from slowing-down to halted and set
    traffic light to red
traffic_light_fsm.add_transition(FSMTransition(slowing_down,
    halted, 't5s', 'R'))

# After 30sec, transition from halted back to free-flowing,
    and set traffic light to green
traffic_light_fsm.add_transition(FSMTransition(halted,
    free_flowing, 't20s', 'G'))

traffic_light_fsm.transitions()
```

An alternative method for defining state transitions in an FSM is by using the `add_transitions_from_function()` method. This approach accepts a callable function that takes two states as arguments: the source state and the target state. The following code demonstrates how this can be implemented.

```
from sage.combinat.finite_state_machine import FSMTransition

# define state transitions, inputs and outputs
def transit_function(state1, state2):
    if state1=='F':
        if state2 =='S':
            return ('t30s', 'Y')

    elif state1=='S':
        if state2 =='H':
            return ('t5s', 'R')

    elif state1=='H':
        if state2 =='F':
            return ('t20s', 'G')

    # all other 'no-transition' combinations
    return None

traffic_light_fsm.add_transitions_from_function(transit_function)
traffic_light_fsm.transitions()
```

Once the states and transitions are defined, the state machine can be run using `process()` method, which then returns the intermediary outputs during the state machine run.

```
# pass in the initial state and the list of inputs
*_, outputs_history = traffic_light_fsm.process(
    initial_state=free_flowing,
    input_tape=['t30s', 't5s', 't20s'],
)

# print out the outputs of the state machine run
outputs_history
```

The `graph()` command displays the graph representation of the state machine.

```
traffic_light_fsm.graph().show(
    figsize=[6, 6],
    vertex_size=800,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0
)
```

The `FiniteStateMachine` class also offers LaTeX representation of the state machine using the `latex_options()` method.

```
# define printout options
traffic_light_fsm.latex_options(
    format_state_label=lambda x: x.label(),
)

# display commands
print(latex(traffic_light_fsm))
```

Note that the LaTeX printout may not have all elements displayed. However, it can still be customized further. The following figure shows a rendering of the above LaTeX commands.

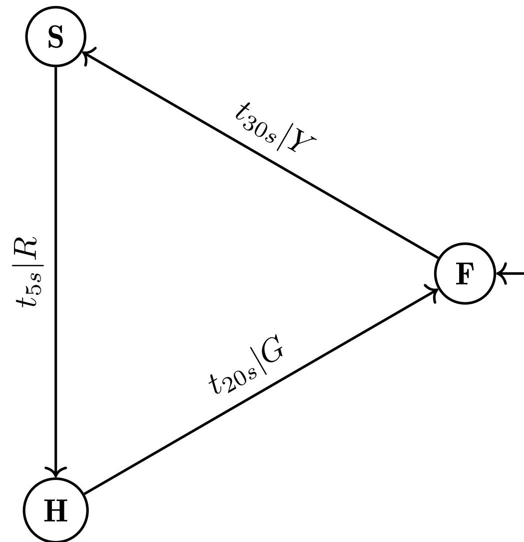

Figure 12.2.3 FSM graph output

12.2.9 Using `Transducer' Module

Sage `Transducer` is a specialization of the generic `FiniteStateMachine` class. The `Transducer` class creates a finite state machine that support optional final states, and whose transitions have input and output labels.

Let's see how to create another state machine using `Transducer` and for the same traffic light example.

```
# the module allows the instantiation of a state machine by
    passing
# the entire state machine definition to the constructor
state_machine_definition = {
    # from-state: [
    #   a list of tuples
    #   (to-state, input, output)
    # ]
    'F': [
        ('F', 't5s',  'G'),
        ('F', 't20s', 'G'),
        ('S', 't30s', 'Y'),
    ],
    'S': [
        ('H', 't5s',  'R'),
        ('S', 't20s', 'Y'),
        ('S', 't30s', 'Y'),
    ],
    'H': [
        ('H', 't5s',  'R'),
        ('F', 't20s', 'G'),
        ('H', 't30s', 'R'),
    ],
}
```

```
traffic_light_transducer = Transducer(
    state_machine_definition,
    initial_states=['F']
)
traffic_light_transducer
```

The member variable `input_alphabet` lists the set of the transducer inputs set.

```
traffic_light_transducer.input_alphabet
```

The member variable `output_alphabet` lists the set of the transducer outputs set.

```
traffic_light_transducer.output_alphabet
```

Since a `Transducer` is also a `FiniteStateMachine` , the method `has_state()` can still be used to check whether or not a given state exists in the defined transducer (by passing in the case-sensitive state label).

```
traffic_light_transducer.has_state('F')
```

The function `states()` enumerates the list of all defined states of the state machine.

```
traffic_light_transducer.states()
```

The method `initial_states()` lists the defined initial state(s) of the state machine.

```
traffic_light_transducer.initial_states()
```

After defining the states and transitions, the transducer can be executed using the `process()` method from the parent `FiniteStateMachine` class. This method returns the intermediate outputs generated during the execution of the state machine.

```
# fetching the initial state by its label
free_flowing = traffic_light_transducer.state('F')

# pass in the initial state and the list of inputs
*_, outputs_history = traffic_light_transducer.process(
    initial_state=free_flowing,
    input_tape=['t30s', 't5s', 't20s'],
)

outputs_history
```

The `graph()` command displays the graph representation of the transducer-based state machine.

Notes. The `latex_options()` method of the base class `FiniteStateMachine` also is inherited and can also be used with `Transducer` state machine to output LaTeX representation.

```
traffic_light_transducer.graph().show(
    figsize=[6, 6],
    vertex_size=800,
```

```
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0
)
```

The above are basic commands with a typical workflow of defining and running of simple finite state machines. The general structure of the state machine can be adapted to fit different use cases. The examples shown can be customized and fine-tuned to reflect more complex scenarios (more states, different input sequences, etc.)

12.3 State Machine in Action

Traffic light systems are crucial for regulating traffic. These systems use carefully coordinated signals to ensure safety for both vehicles and pedestrians. In the previous section, the traffic light system was modeled in an overly simplistic way. This section adds complexity to account for pedestrian presence, ensuring safe crossings while maintaining smooth traffic flow.

12.3.1 Traffic Light Controller: Problem Overview

Let's design a traffic light system for a two-way road with pedestrian crossings. This system coordinates the movement of vehicles and pedestrians using lights to indicate when vehicles can proceed, slow down, or stop, and when pedestrians can cross safely. Vehicle traffic lights include three signals: Red, Yellow, and Green. For simplicity, the pedestrian lights also use three signals: red, yellow, and green. Signal transitions are governed by timers, as described in the previous section, with each timer triggering a transition event after a predefined duration.

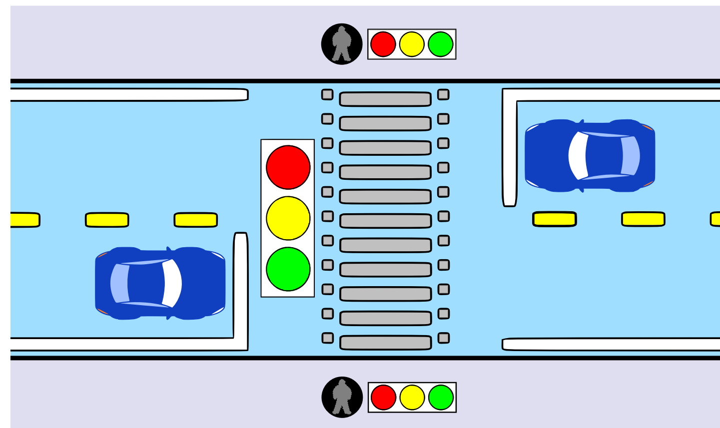

Figure 12.3.1 Simple Traffic Light

The system must ensure safety and smooth traffic flow by coordinating appropriate traffic and pedestrian light configurations. Initially, vehicle traffic proceeds with a traffic green light, while pedestrian crossing is prohibited with a pedestrian red light.

12.3.2 Elements of the FSM Model

The goal here is to define a state machine model that can control this traffic light system, construct it, then put it under test. This system has different configurations of lights: Red (`R`), Yellow (`Y`), and Green (`G`) for traffic, and red (`r`), yellow (`y`), and green (`g`) for pedestrians. Note that not all possible combinations makes sense.

For inputs, the system leverage four independent timers with different preset durations and triggering different use cases as follows:

- 30sec timer $t30s$: drives the traffic light transition from `G` to `Y`. The pedestrian light remains `r` and unchanged.
- 5sec timer $t5s$: drives the traffic light transition from `Y` to `R`, and the pedestrian light transition from `r` to `g`.
- 20sec timer $t20s$: drives the pedestrian light transition from `g` to `y`, while the traffic light remains `R` and unchanged.
- 10sec timer $t10s$: drives the pedestrian light transition from `y` to `r`, and the traffic light transition from `R` back to `G`.

From the above timers and lights configurations, the following set of 4 distinct states emerges:

- State `Yr`: Where the traffic light is Yellow, pedestrian light is red.
- State `Rg`: Where the traffic light is Red, pedestrian light is green.
- State `Ry`: Where the traffic light is Red, pedestrian light is yellow.
- State `Gr`: Where the traffic light is Green, pedestrian light is red.

Finally, the system's outputs corresponding to each of the above are the light configurations and would be similar to the states:

- `(Y,r)`: Traffic light turning Yellow and the pedestrian light remains red.
- `(R,g)`: Traffic light turning Red and the pedestrian light turning green.
- `(R,y)`: Traffic light remains Red and the pedestrian light turning yellow.
- `(G,r)`: Traffic light turning Green and the pedestrian light turning red.

The following table summarize the elements of the new traffic light FSM.

Table 12.3.2 The Traffic Light State Machine Definition

	next				output			
current	t_{5s}	t_{10s}	t_{20s}	t_{30s}	t_{5s}	t_{10s}	t_{20s}	t_{30s}
Gr	–	–	–	Yr	–	–	–	(Y, r)
Yr	Rg	–	–	–	(R, g)	–	–	–
Rg	–	–	Ry	–	–	–	(R, y)	–
Ry	–	Gr	–	–	–	(G, r)	–	–

The symbol – indicates no state change, and no output change.

12.3.3 Construct the FSM

```
# FSM elements
states = ['Gr', 'Yr', 'Rg', 'Ry']
inputs = ['t5s', 't10s', 't20s', 't30s']
outputs = ['(G,r)', '(Y,r)', '(R,g)', '(R,y)']

# Traffic light state machine definition
significant_configs = [
    # from-state, to-state, input, output
    ('Gr', 'Yr', 't30s', '(Y,r)'),
    ('Yr', 'Rg', 't5s',  '(R,g)'),
    ('Rg', 'Ry', 't20s', '(R,y)'),
    ('Ry', 'Gr', 't10s', '(G,r)'),
]

machine_configs = {}
for config in significant_configs:
    (fr, to, evt, out) = config
    # Add the significant transition
    machine_configs[fr] = [
        (to, evt, out),
    ]
    # Add no-state change transitions
    machine_configs[fr].extend(
        [(fr, event, '_') for event in inputs if event !=
            evt]
    )

traffic_light_controller = FiniteStateMachine(
    machine_configs,
    initial_states=['Gr']
)

print('States:', traffic_light_controller.states())
print()
[print(_) for _ in traffic_light_controller.transitions()]
print()

print('-'*100)
print('FSM␣Config:', machine_configs)
traffic_light_controller
```

12.3.4 Display the State Transition Graph

The FSM is visualized as before (a directed graph with nodes representing states and edges showing transitions).

```
traffic_light_controller.graph().plot(
    figsize=[6, 6],
    vertex_size=800,
    edge_labels=True,
    vertex_labels=True,
    edge_color=(.2,.4,1),
    edge_thickness=1.0
)
```

12.3.5 Simulate a Full Cycle Run of the FSM

The simulation starts in the initial state (`Gr`) and transitions through all states.

```
# pass in the initial state and the list of inputs
Gr = traffic_light_controller.state('Gr')
*_, outputs_history = traffic_light_controller.process(
    initial_state=Gr,
    input_tape=['t30s', 't5s', 't20s', 't10s', 't30s'],
)

# print out the outputs of the state machine run
print("FSM outputs:")
[print(_) for _ in outputs_history];
print()
```

It is worth noting that using Sage built-in modules could produce an error when handling transitions that were not defined in the FSM. For instance, in the previous example, if the timer durations __pattern__ for the input does not match the defined transitions, the output will be a `None` value. Similarly, an exception would be thrown if attempting to run the FSM starting at state that is not part of the FSM definition.

References

We based most of this text on the Discrete Math lectures at Wilbur Wright College, taught by Professor Hellen Colman. We focused our efforts on creating original work, and we drew inspiration from the following sources:

Doerr, Al, and Ken Levasseur. ADS Applied Discrete Structures. https://discretemath.org, 21 May 2023.SageMath, the Sage Mathematics Software System (Version 10.2), The Sage Developers, 2024, https://www.sagemath.org.Beezer, Robert A., et al. The PreTeXt Guide. Pretextbook.org, 2024, https://pretextbook.org/doc/guide/html/guide-toc.html.Zimmermann, Paul. Computational Mathematics with SageMath. Society For Industrial And Applied Mathematics, 2019.

Colophon

This book was authored in PreTeXt.

Index

antisymmetric, 32
arithmetic operators, 1
assignment operator, 17

binomial, 24
binomial coefficients, 42
bipartite, 53
Boolean Algebra, 76
Boolean functions, 77
Breadth-first search, 63

cardinality, 18
combination, 24
connected, 52
contradiction, 28
cycle, 62

data types
  boolean, 4
  dictionary, 5
  integer, 3
  list, 4
  rational, 4
  set (Python), 4
  Set (Sage), 17
  string, 4
  symbolic, 3
  tuple, 4
debugging
  attribute error, 13
  logical error, 13
  name error, 12
  syntax error, 12
  type error, 13
  value error, 13
diameter, 52
digraph, 30
dodecahedron, 57

equality operator, 17
equivalence, 34
error message, 12
Euler circuit, 55
Euler path, 55

factorial, 24
Fibonacci, 41
forest, 62
functions (math), 40
functions (programming), 7

graph plotting
  Color, 47
  edge color, 48
  edge style, 51
  figsize, 47
  layout, 50
  vertex size, 47
graphs
  add edge, 46
  add vertex, 46
  adjacency matrix, 45
  arcs, 43
  degree, 45
  delete vertex, 46
  edges, 45
  graph definition, 43
  incidence matrix, 46
  links, 43
  nodes, 43
  order, 45
  remove vertex, 46
  size, 45
  vertices, 44
  weighted, 44

Hamilton circuit, 57
Hamilton path, 57
Hamiltonian cycle, 57
Hasse diagram, 35

identifiers, 3

isomorphism, 53
iteration
  for loop, 6
  list comprehension, 6

join, 73
join matrix, 74

lattice, 72
logical operators
  and, 26
  biconditional, 26
  conditional, 26
  not, 26
  or, 26

meet, 73
meet matrix, 73
minimal spanning tree, 64

NP-hard, 57

partial order, 35
path, 52
permutation, 25
poset, 35
Prim's algorithm, 64

recursion, 40
reflexive, 31
relation, 29
run code
  CoCalc, 15
  Jupyter Notebook, 15
  local, 15
  SageMath worksheets, 15

search algorithms, 63
sequence, 41
set operations
  Cartesian product, 21
  compliment, 20
  difference, 20
  intersection, 19
  power set, 21
  union, 19
spanning subgraph, 63
spanning tree, 63
state machines
  application, 98
  definition, 87
  model, 89
subgraph, 63
symmetric, 32

tautology, 27
transitive, 33
traveling salesperson, 57
tree, 62
truth table, 26