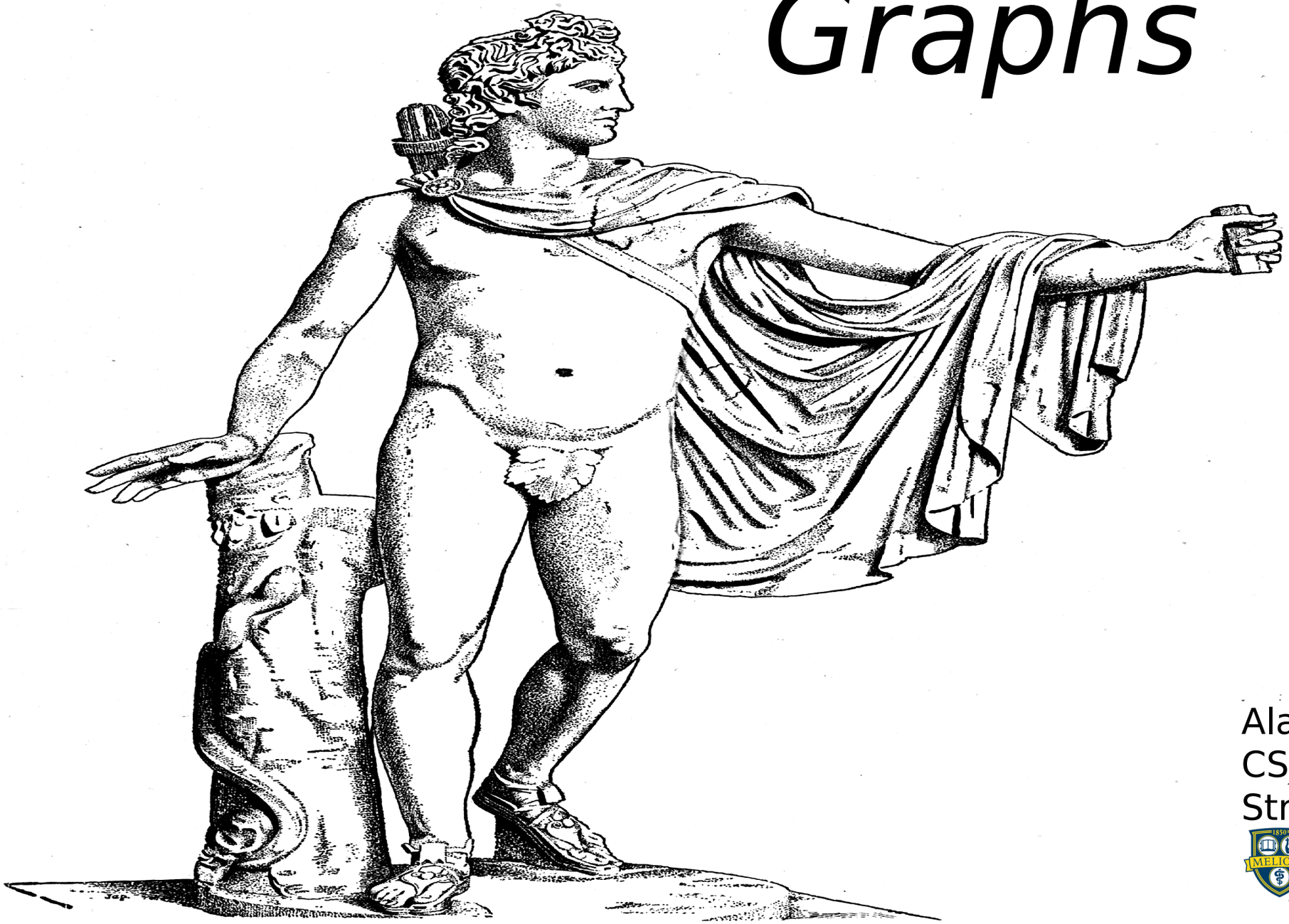
The Art of Data Structures

Graphs



Alan Beadle

CSC 162: The Art of Data Structures

UNIVERSITY of ROCHESTER

Agenda

- To learn what a graph is and how it is used
- To implement the graph abstract data type using multiple internal representations
- To see how graphs can be used to solve a wide variety of problems

Graphs

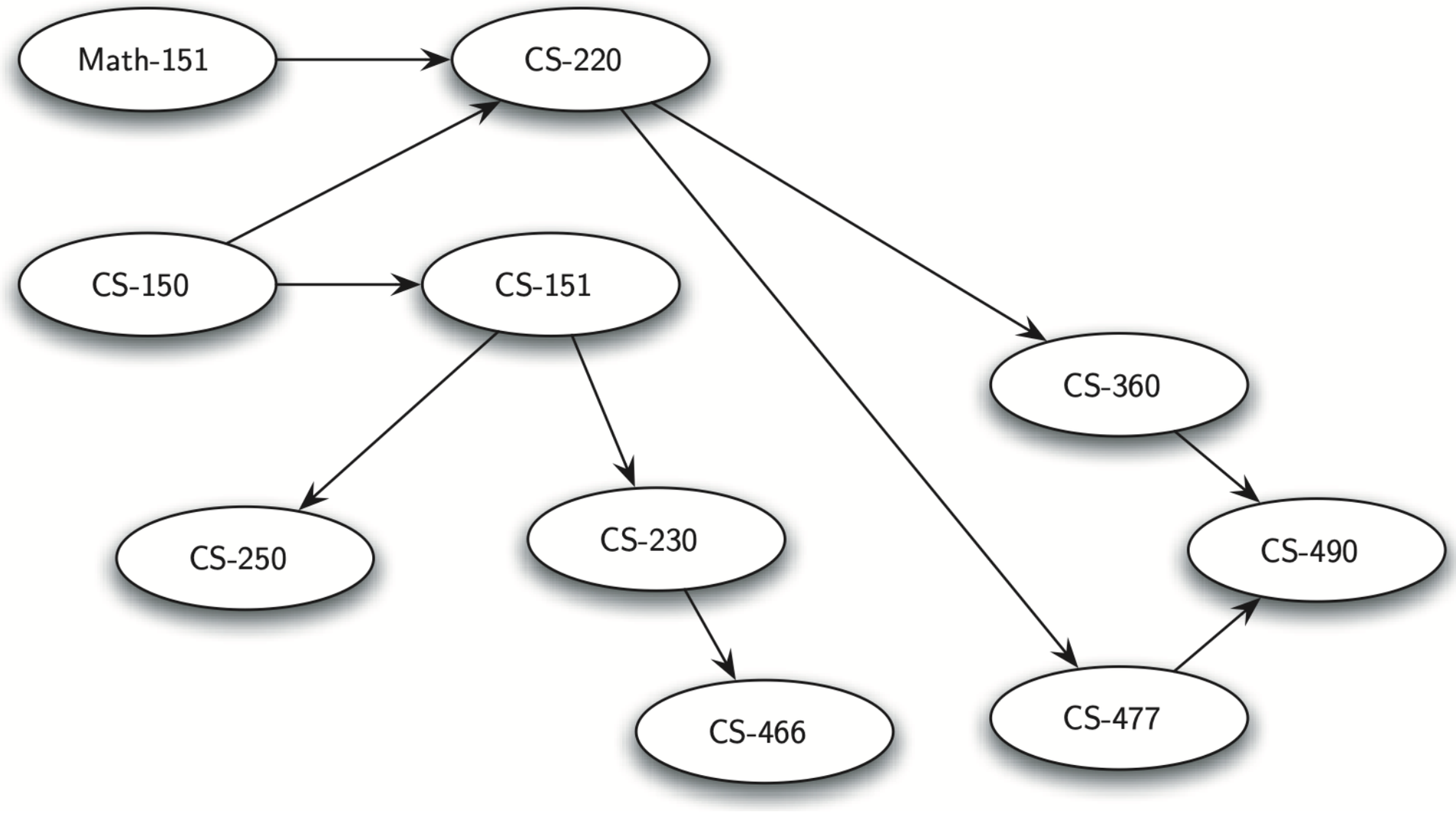
Can be:

- Directed (vs undirected)
- Acyclic (vs cyclic)
- Weighted (vs unweighted)

There are many other properties too.

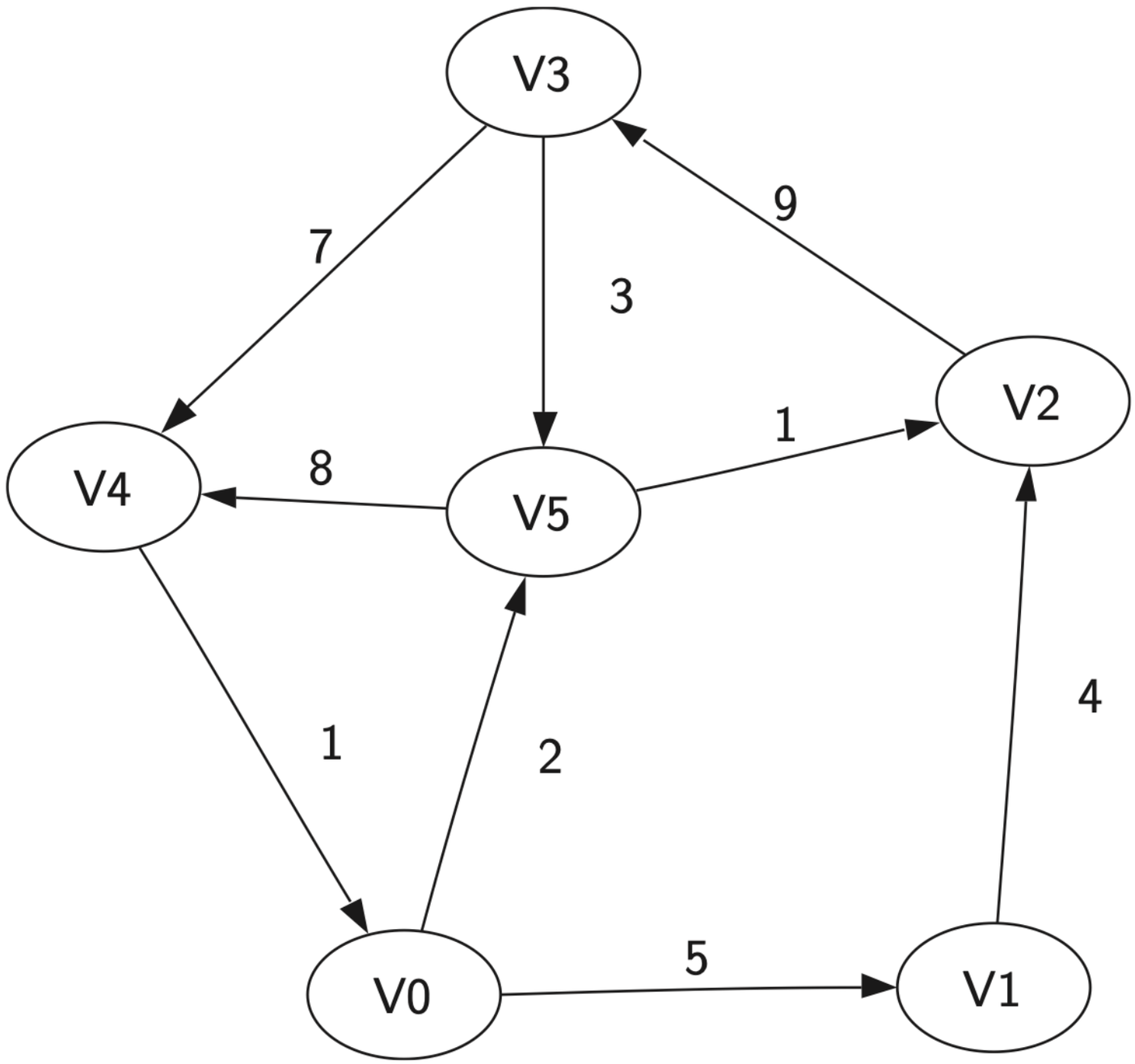
Graphs

Prerequisites for a Computer Science Major



Graphs

A Simple Example of a Directed (and weighted) Graph



Graphs
Specification

- `Graph()` creates a new, empty graph
- `add_vertex(vert)` adds an instance of Vertex to the graph
- `add_edge(from_vert, to_vert)` Adds a new, directed edge to the graph; connects two vertices
- `get_vertex(vert_key)` finds the vertex in the graph named vertKey
- `get_vertices()` returns the list of all vertices in the graph

Representation

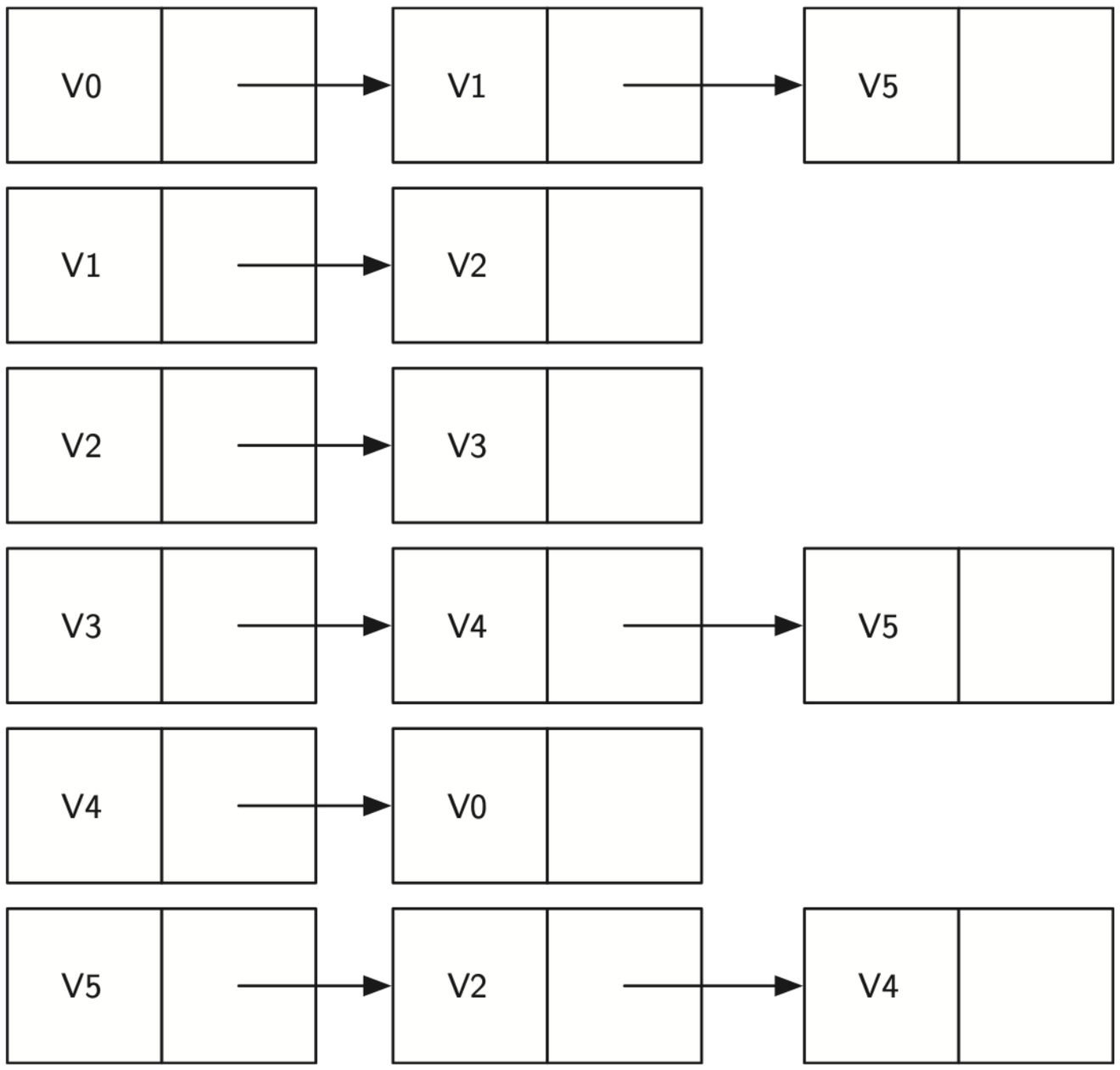
Representation

Adjacency Matrix

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

Representation

Adjacency List



Implementation

Implementation

The `Graph` *Class*

```
from Vertex import Vertex

class Graph:
    def __init__(self):
        self.vert_list = {}
        self.num_vertices = 0

    def add_vertex(self, key):
        self.num_vertices = self.num_vertices + 1
        new_vertex = Vertex(key)
        self.vert_list[key] = new_vertex
        return new_vertex

    def getVertex(self, n):
        if n in self.vert_list:
            return self.vert_list[n]
        else:
            return None
```

Implementation

The `Graph` *Class (cont.)*

```
def __contains__(self,n):
    return n in self.vert_list

def add_edge(self, f, t, cost=0):
    if f not in self.vert_list:
        nv = self.add_vertex(f)
    if t not in self.vert_list:
        nv = self.add_vertex(t)
    self.vert_list[f].add_neighbor(self.vert_list[t], cost)

def get_vertices(self):
    return self.vert_list.keys()

def __iter__(self):
    return iter(self.vert_list.values())
```

Implementation

The Vertex *Class*

```
class Vertex:
    def __init__(self, key):
        self.id = key
        self.connected_to = {}
        self.color = 'white'
        self.dist = sys.maxsize
        self.pred = None
        self.disc = 0
        self.fin = 0

    def add_neighbor(self, nbr, weight=0):
        self.connected_to[nbr] = weight

    def set_color(self, color):
        self.color = color

    def set_distance(self, d):
        self.dist = d
```

Implementation

The `Vertex` *Class (cont.)*

```
def set_pred(self, p):
    self.pred = p

def set_discovery(self, dtime):
    self.disc = dtime

def set_finish(self, ftime):
    self.fin = ftime

def get_finish(self):
    return self.fin

def get_discovery(self):
    return self.disc

def get_pred(self):
    return self.pred
```

Implementation

The `Vertex` *Class (cont.)*

```
def get_distance(self):
    return self.dist

def get_color(self):
    return self.color

def get_connections(self):
    return self.connected_to.keys()

def get_weight(self, nbr):
    return self.connected_to[nbr]

def __str__(self):
    return str(self.id) + ' connected_to: '\
        + str([x.id for x in self.connected_to])

def get_id(self):
    return self.id
```

Implementation

Test

```
from Graph import Graph

g = Graph()
for i in range(6):
    g.add_vertex(i)
print(g.vert_list)

g.add_edge(0,1,5)
g.add_edge(0,5,2)
g.add_edge(1,2,4)
g.add_edge(2,3,9)
g.add_edge(3,4,7)
g.add_edge(3,5,3)
g.add_edge(4,0,1)
g.add_edge(5,4,8)
g.add_edge(5,2,1)

for v in g:
    for w in v.get_connections():
        print("( %s , %s )" % (v.get_id(), w.get_id()))
```

Questions?