# Graphs

## Introduction and Breadth First Search

Brad Miller  David Ranum[1]

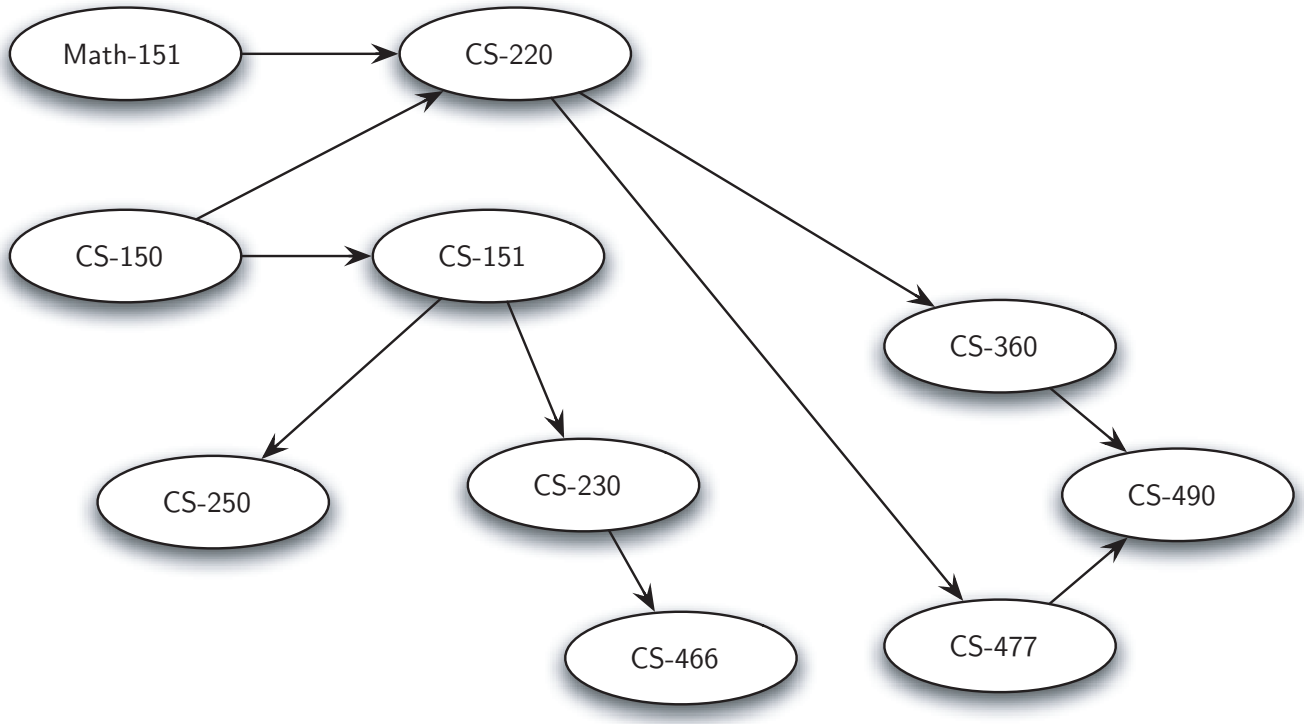
[1]Department of Computer Science
Luther College

12/19/2005

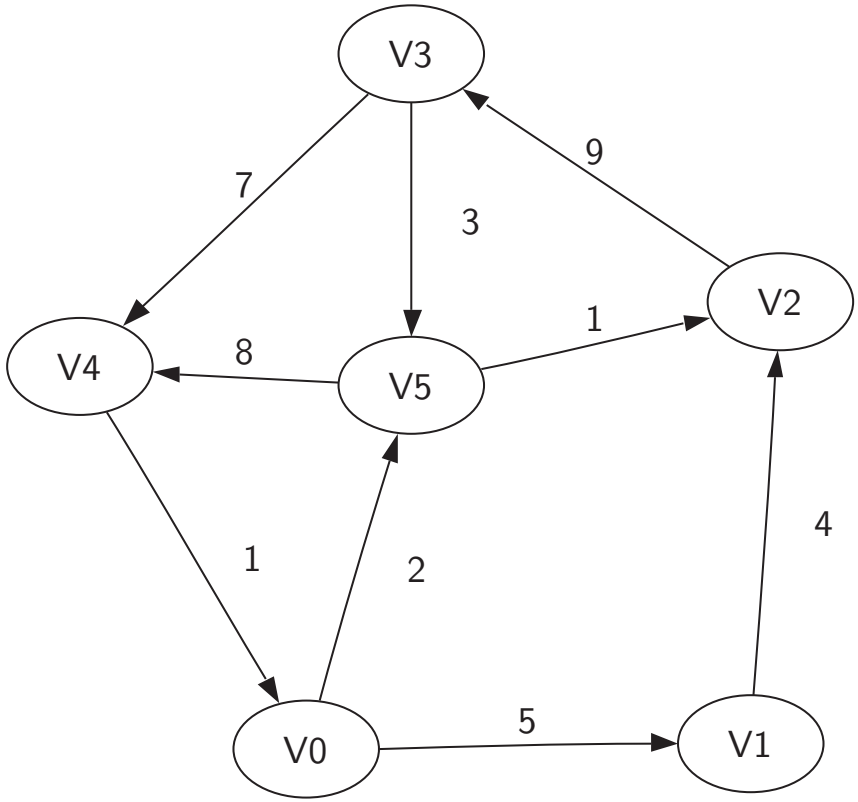
# Outline

- 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

# Prerequisites for a Computer Science Major



# A Simple Example of a Directed Graph



- `Graph()` creates a new, empty graph.
- `addVertex(vert)` adds an instance of `Vertex` to the graph.
- `addEdge(fromVert, toVert)` Adds a new, directed edge to the graph that connects two vertices.
- `getVertex(vertKey)` finds the vertex in the graph named `vertKey`.
- `getVertices()` returns the list of all vertices in the graph.

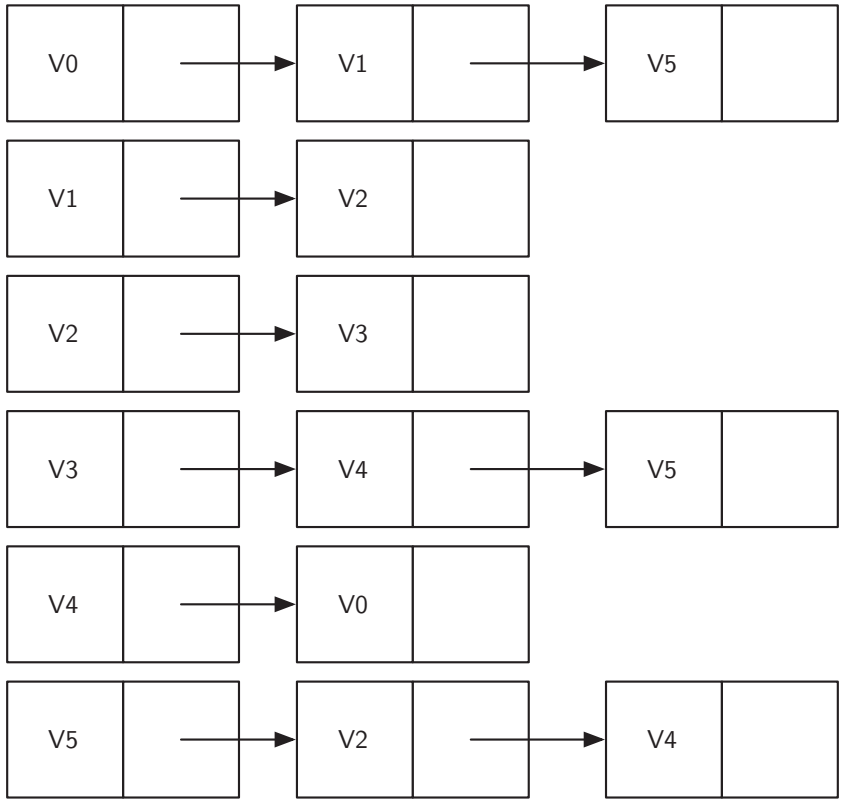
# Outline

# An Adjacency Matrix Representation for a Graph

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

# Outline

# An Adjacency List Representation of a Graph



# Outline

# The `Vertex` Class I

```
1  class Vertex:
2      def __init__(self,num):
3          self.id = num
4          self.adj = []
5          self.color = 'white'
6          self.dist = sys.maxint
7          self.pred = None
8          self.disc = 0
9          self.fin = 0
10         self.cost = {}
11
12     def addNeighbor(self,nbr,cost=0):
13         self.adj.append(nbr)
14         self.cost[nbr] = cost
15
16     def __str__(self):
```

# The `Vertex` Class II

```
17            return str(self.id) + ":color " + self.color + \
18                   ":dist " + str(self.dist) +
19                   ":pred [" + str(self.pred)+ "]\n"
20
21        def getCost(self,nbr):
22            return self.cost[nbr]
23        def setCost(self,nbr,cost):
24            self.cost[nbr] = cost
25        def setColor(self,color):
26            self.color = color
27        def setDistance(self,d):
28            self.dist = d
29        def setPred(self,p):
30            self.pred = p
31        def setDiscovery(self,dtime):
32            self.disc = dtime
33        def setFinish(self,ftime):
```

# The `Vertex` Class III

```
34          self.fin = ftime
35      def getFinish(self):
36          return self.fin
37      def getDiscovery(self):
38          return self.disc
39      def getPred(self):
40          return self.pred
41      def getDistance(self):
42          return self.dist
43      def getColor(self):
44          return self.color
45      def getAdj(self):
46          return self.adj
47      def getId(self):
48          return self.id
```

# The `Graph` Class I

```
1  class Graph:
2      def __init__(self):
3          self.vertList = {}
4          self.numVertices = 0
5
6      def addVertex(self,key):
7          self.numVertices = self.numVertices + 1
8          newVertex = Vertex(key)
9          self.vertList[key] = newVertex
10         return newVertex
11
12     def getVertex(self,n):
13         if self.vertList.has_key(n):
14             return self.vertList[n]
15         else:
16             return None
```

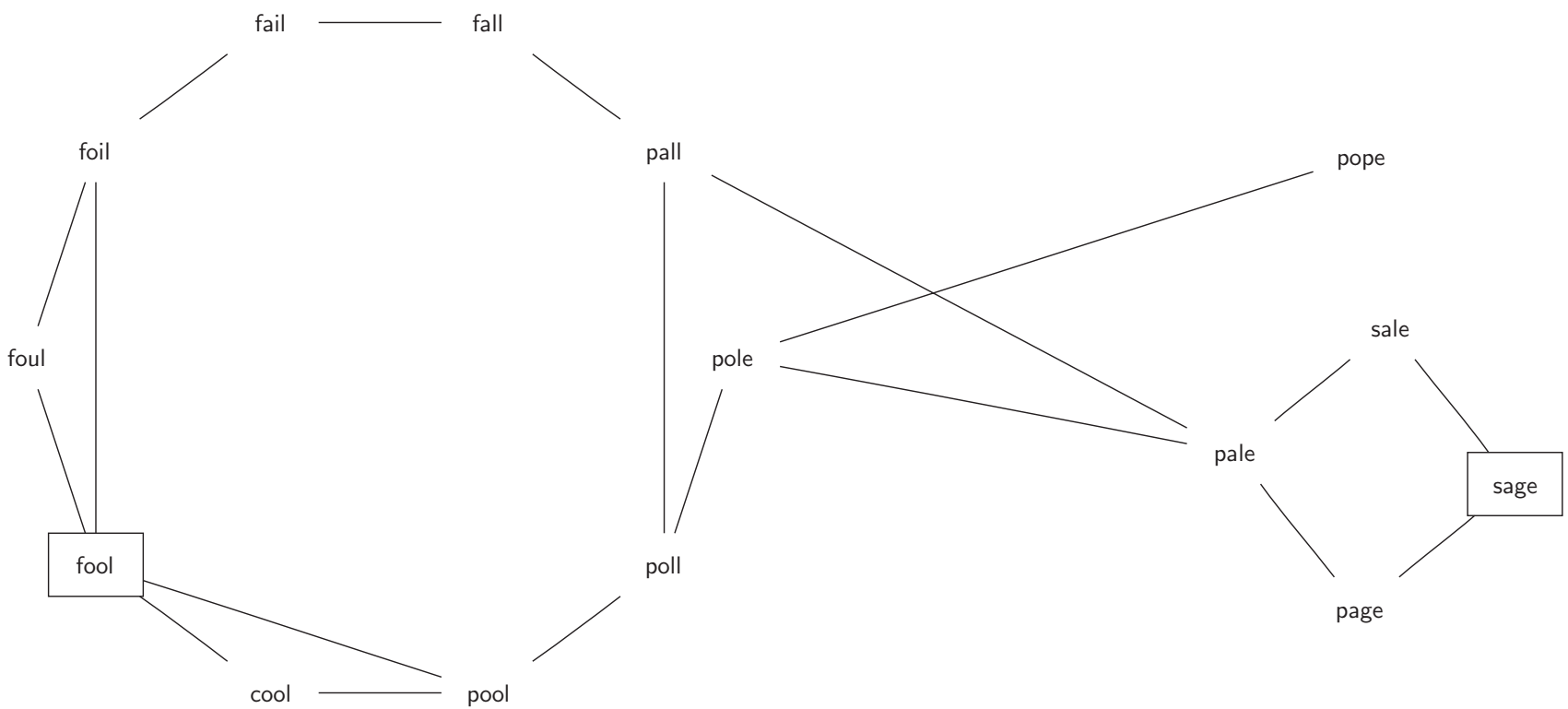
# The `Graph` Class II

```
17
18      def has_key(self,n):
19          return self.vertList.has_key(n)
20
21      def addEdge(self,f,t,c=0):
22          if not self.vertList.has_key(f):
23              nv = self.addVertex(f)
24          if not self.vertList.has_key(t):
25              nv = self.addVertex(t)
26          self.vertList[f].addNeighbor(self.vertList[t],c)
27
28      def getVertices(self):
29          return self.vertList.values()
30
31      def __iter__(self):
32          return self.vertList.itervalues()
```

# Outline

- Represent the relationships between the words as a graph.
- Use the graph algorithm known as breadth first search to find an efficient path from the starting word to the ending word.

# A Small Word Ladder Graph



# Building a Graph of Words for the Word Ladder Problem I

```
1  def buildGraph():
2      d = {}
3      g = Graph()
4      wfile = file('words.dat')
5      # create buckets of words that differ by one letter.
6      for line in wfile:
7          word = line[0:5]
8          for i in range(5):
9              bucket = word[0:i] + '_' + word[i+1:5]
10             if d.has_key(bucket):
11                 d[bucket].append(word)
12             else:
13                 d[bucket] = [word]
14     # add vertices and edges for words in the same bucket.
```

# Building a Graph of Words for the Word Ladder Problem II

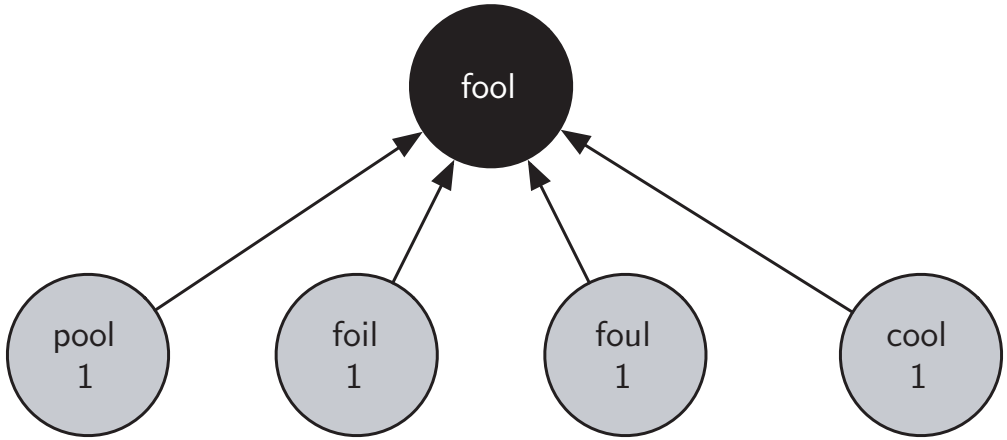
```
15        for i in d.keys():
16            for j in d[i]:
17                for k in d[i]:
18                    if j != k:
19                        g.addEdge(j,k)
20        return g
```

1. The new, unexplored vertex `v`, is colored gray.
2. The predecessor of `v` is set to the current node `w`
3. The distance to `v` is set to the distance to `w + 1`
4. `v` is added to the end of a queue. Adding `v` to the end of the queue effectively schedules this node for further exploration, but not until all the other vertices on the adjacency list of `w` have been explored.

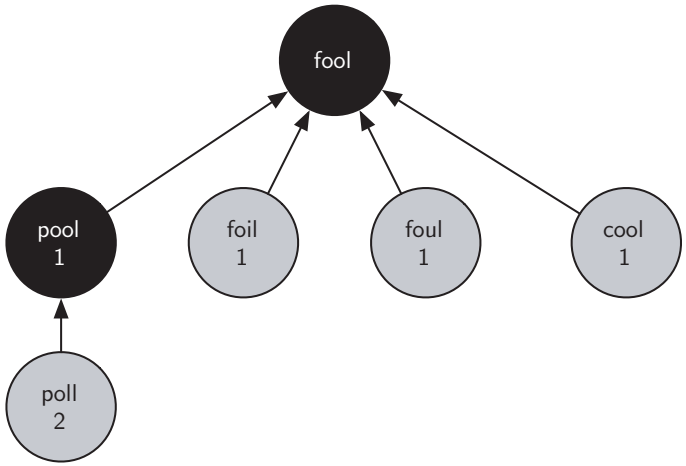
# Breadth First Search I

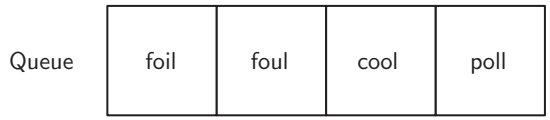
```
1  def bfs(g,vertKey):
2      s = g.getVertex(vertKey)
3      s.setDistance(0)
4      s.setPred(None)
5      s.setColor('gray')
6      Q = Queue()
7      Q.enqueue(s)
8      while (Q.size() > 0):
9          w = Q.dequeue()
10         for v in w.getAdj():
11             if (v.getColor() == 'white'):
12                 v.setColor('gray')
13                 v.setDistance( w.getDistance() + 1 )
14                 v.setPred(w)
15                 Q.enqueue(v)
16         w.setColor('black')
```

# Fist Step in the Breadth First Search

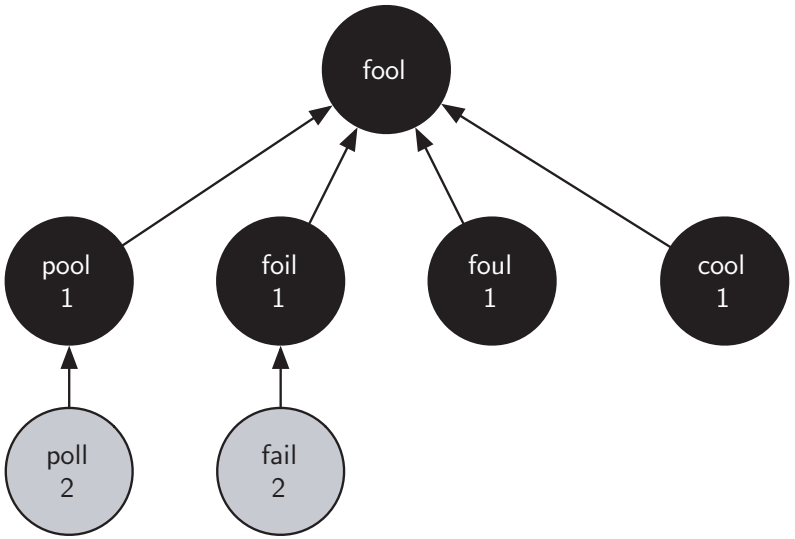


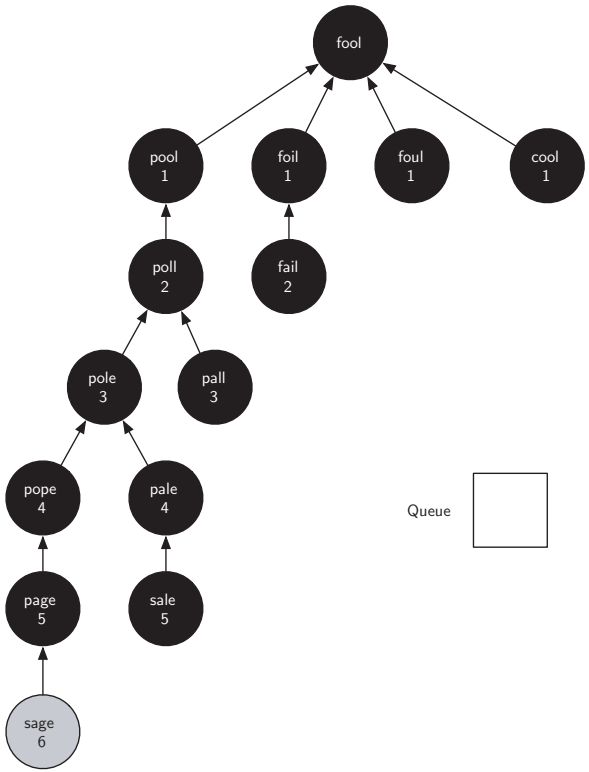

# The Second Step in the Breadth First Search



# Constructing the Breadth First Search Tree




(a) Breadth First Search Tree After Completing One Level

(b) Final Breadth First Search Tree

- You can represent your problem in terms of an unweighted graph.
- The solution to your problem is to find the shortest path between two nodes in the graph.