Overview

• Intro to graphs
• Abstract data type (ADT)
• Adjacency matrix
• Word Letter Puzzle
• Breadth First Search
Objective

• Learn what a graph is and how it is used
• Able 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

• More general structure than trees
  • Tree is a special type of graph
• Graphs can represent:
  • Roads
  • Airline flights from city to city
  • How the Internet is connected
• Once good representation for problem, graph algorithms can be applied
Graphs

- Computer can understand roadmap as a graph
- Computer can use graph representation to determine:
  - Shortest path
  - Easiest path
  - Quickest path
Graphs - Definitions

• **Vertex (node):**
  - Fundamental part of a graph.
  - Can have a name => “key.”
  - Additional information as “payload.”

• **Edge:**
  - Connects two vertices
  - One-way or two-way
  - One-way => directed graph (as in example on last slide)

• **Weight:**
  - Edges may be weighted
  - Cost to go from one vertex to another
  - E.g., cost on edges of graph of roads might represent distance between two cities
Formally Define Graph

• Graph can be represented by $G$, where $G=(V,E)$
• $V$ is a set of vertices, and $E$ is a set of edges
• Each edge is a set of tuples where $w, v \in V$
• Weight can be added as third component of tuple
• A subgraph is a set of edges $e$ and vertices $v$ such that $e \subseteq E$ and $v \subseteq V$

Graph Example

$E = \{(v0, v1, 5), (v1, v2, 0), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1), (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1)\}$

$V = \{v0, v1, v2, v3, v4, v5\}$
**Graphs - Definitions**

- **Path:**
  - \( w_1, w_2, \ldots, w_n \), such that \((w_i, w_{i+1}) \in E\) for all \(1 \leq i \leq n-1\)
  - Path length = number of edges in path => \(n-1\)

- **Cycle:**
  - Path that starts and ends at same vertex
  - \((V5, V2, V3, V5)\)
  - Acyclic graph => no cycles
  - Directed acyclic graph (DAG) => directed graph with no cycles

---

**Graphs – Abstract Data Type**

- **Graph()** creates 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
- **addEdge(fromVert, toVert, weight)** adds a new, weighted, directed edge to the graph that connects two vertices
Graphs – Abstract Data Type

- `getVertex(vertKey)` finds the vertex in the graph named `vertKey`
- `getVertices()` returns the list of all vertices in the graph
- `in` returns `True` for a statement of the form `vertex in graph`, if the given vertex is in the graph, `False` otherwise

Graphs – Adjacency Matrix

```
V0  V1  V2  V3  V4  V5
V0  5   -   3   -   2
V1  4   -   -   -   -
V2  -   9   -   -   -
V3  -   -   -   7   3
V4  1   -   -   1   -
V5  1   -   -   -   8
```
**Graphs – Adjacency Matrix**

- For small graphs:
  - Easy to see which nodes are connected
  - But matrix is sparse (many empty cells), not efficient
- Adjacency matrix is good fit when #edges is large
  - Completely filled matrix would require to connect all edges with each other -> doesn’t occur often

---

**Adjacency List**

- Master list of all vertices
- Each vertex maintains list of other vertices its connected to
- Vertex class uses dictionary instead of list
- Keys are vertices and values are weights
- Benefits:
  - Compactly represent sparse matrix
  - Easily find links
Vertex Class

- Each Vertex uses dictionary to keep track of the vertices to which it is connected, and the weight of each edge
- dictionary is called connectedTo
- addNeighbor method is used to add a connection from this vertex to another
- getConnections method returns all of the vertices in the adjacency list
- getWeight method returns the weight of the edge

```python
class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

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

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self, nbr):
        return self.connectedTo[nbr]
```
Graphs Class

• Contains dictionary that maps vertex names to vertex objects
• Class also provides methods for adding and connecting vertices
• `getVertices` returns names of all vertices in graph
• `__iter__` allows iteration over particular graph

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0
    def addVertex(self, key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex
    def getVertex(self, n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None
    def __contains__(self, n):
        return n in self.vertList
    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)
    def getVertices(self):
        return self.vertList.keys()
    def __iter__(self):
        return iter(self.vertList.values())
Graph Class - Operations

```python
>>> g = Graph()
for i in range(6):
    g.addVertex(i)
>>> g.vertList
['0', '1', '2', '3', '4', '5']
>>> g.addEdge(0, 1, 5)
>>> g.addEdge(1, 2, 4)
>>> g.addEdge(2, 3, 9)
>>> g.addEdge(3, 4, 7)
>>> g.addEdge(3, 5, 3)
>>> g.addEdge(4, 0, 1)
>>> g.addEdge(5, 4, 8)
>>> g.addEdge(5, 2, 1)
for v in g:
    for w in v.getConnections():
        print(('%s %s' % (v.getId(), w.getId())))
```

Word Letter Puzzle

• Goal: Transform word “FOOL” into ”SAGE”
  • Change one letter at a time
  • At each step: transform one word into another

- FOOL
- POOL
- POLL
- POLE
- PALE
- SALE
- SAGE
Word Letter Puzzle

• Can solve this problem using a graph algorithm
  • Represent relationships between words as graph
  • Use breadth first search algorithm
    • Finds efficient path from starting work to ending word

Word Letter Puzzle

• First problem: how to turn large collection of words into a graph
  • Only connect words that differ by single letter
  • If such graph can be created, any path from one word to another is a solution
Word Letter Puzzle

Tackling the Problem

• Let's assume list of words, all same length
  1. Starting point: Create a vertex for every word in list
  2. Compare all words with each other
  3. If different by one letter => create edge between them
Tackling the Problem

Analysis:

- Assume list of 5,110 words
- Comparing one word to each other is $\approx O(n^2)$
- For 5,110 words that is more than 26 million comparisons

Improved Approach

- Huge # of bucket
- Each with 4-letter word on top
- One letter is wildcard “_”
- Example: “POPE” and ”POPS” match “POP_”
- When matching bucket is found, add word
- Once all words in right bucket => must be connected in graph
Building the Graph

```python
from pythonds.graphs import Graph

def buildGraph(wordFile):
    d = {}
    g = Graph()
    wfile = open(wordFile,'r')
    # create buckets of words that differ by one letter
    for line in wfile:
        word = line[:-1]
        for i in range(len(word)):
            bucket = word[:i] + '_' + word[i+1:]
            if bucket in d:
                d[bucket].append(word)
            else:
                d[bucket] = [word]
    # add vertices and edges for words in the same bucket
    for bucket in d.keys():
        for word1 in d[bucket]:
            for word2 in d[bucket]:
                if word1 != word2:
                    g.addEdge(word1,word2)
    return g
```

Sparsity of Matrix

Analysis:

- 5,110 four-letter words
- Adjacency matrix would have $5,110^2 = 26,112,100$ cells
- Graph created by `buildGraph()` has 53,286 edges
- => Only .2% of matrix cells would be filled!
Implementing Breadth First Search

- Breadth First Search (BFS) is one of the easiest algorithms to search a graph
- Given a graph $G$ and starting vertex $s$, BFS explores edges in the graph to find all vertices for which there is a path from $s$.
- Note: BFS finds all vertices at distance $k$ from $s$, before any vertices at distance $k+1$.

Implementing Breadth First Search

- To visualize BFS, imagine that it is building one level at a time.
- BFS adds all children of the starting vertex.
- Then it begins to discover any of the grandchildren.
- To keep track of progress, edges are colored white, gray, or black:
  - White: undiscovered vertex
  - Gray: initially discovered
  - Black: vertex is colored black when completely explored
• Starting with fool, add all nodes adjacent to it
• Added as new nodes to expand
BFS Example

- Removes "pool" from front of the queue
- Repeats process for "pool"
- When "cool" is examined alg. detects that it is already grey => shorter path to cool already exists
- "poll" is only new node added

BFS Example

- Next word in queue is "foil"
- Only new node that "foil" can add is "fail"
- Neither of next two nodes add anything new to queue or tree
- Figure shows tree after expanding all vertices on 2\textsuperscript{nd} level
**BFS Example**

- Final BFS tree shown
- With BFS tree, can start at any vertex and follow predecessor arrows back to
- Find shortest word ladder from any word in the tree back to the starting vertex

---

**Building the Graph**

```python
def traverse(y):
    x = y
    while (x.getPred()):
        print(x.getId())
        x = x.getPred()
        print(x.getId())

traverse(g.getVertex('sage'))
```

- Function `traverse()` shows how to follow the predecessor links to print out the word ladder
Next Steps

• Next lecture on Thursday: Breadth First Search, Depth First Search
• Next discussion on Thursday: Graphs and BFS
• Project 1 due on Thursday 10/25 at 11PM