Overview

- Breadth First Search
- Depth First Search
**Objective**

- Understand and be able to apply the breadth first search (BFS) algorithm
- Understand and be able to apply the depth first search (DFS) algorithm

---

**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
Tackling the Problem

- Lets 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 BFD, 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
**BFS Example**

- 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 2nd 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

```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
**BFS - Analysis**

- While loop executed at least one time per vertex => $O(V)$
- Loop nested within while loop is at least executed once for each edge in graph $O(E)$
- In combination: $O(E+V)$
- Following link from starting node to goal node => $O(V)$
- Plus time required to build initial graph

**Shortest Path Problems**

- When you surf the web, send email, or chat, lots of work is going on behind the scenes
- Specific details on how computer networks and the Internet work, is part of a different class
- However, we’ll learn just enough to understand another very important graph algorithm
Shortest Path Problems

- When web page is requested from server, request travels over local network on to the Internet via a router
- Request travels over Internet and eventually arrives at router that connect local area network where web server is located
- Web page travels same route back to client

traceroute to www.heise.de (193.99.144.85), 64 hops max, 52 byte packets
1  128.119.232.1 (128.119.232.1)  5.001 ms  1.859 ms  1.879 ms
2  core1-rxe-0-1-1.gw.umass.edu (128.119.0.237)  1.843 ms  1.929 ms  1.811 ms
3  border1-rt-et-5-0-0.gw.umass.edu (192.80.83.102)  1.817 ms  2.185 ms
border2-rt-et-5-0-0.gw.umass.edu (192.80.83.110)  2.013 ms
4  5-2-19.ear3.newyork1.level3.net (4.71.230.233)  7.528 ms
border2-rt-et-4-0-1.gw.umass.edu (192.80.83.97)  2.042 ms  2.105 ms
5  5-2-19.ear3.newyork1.level3.net (4.71.230.233)  7.413 ms
ae-1-3110.edge4.frankfurt1.level3.net (4.69.163.100)  84.297 ms
5-2-19.ear3.newyork1.level3.net (4.71.230.233)  7.824 ms
6  ae-1-3110.edge4.frankfurt1.level3.net (4.69.163.106)  84.635 ms  84.664 ms  84.442 ms
7  te2-2.c102.f.de.plusline.net (212.162.24.58)  86.318 ms  87.387 ms
8  * * *
9  212.19.61.13 (212.19.61.13)  85.865 ms !X * *
10 * * 212.19.61.13 (212.19.61.13)  85.324 ms !X

Shortest Path Problems

- Many additional routers are inside Internet “cloud”
- Routers work together to get information (data packets) from place to place
Shortest Path Problems

traceroute to www.heise.de (193.99.144.85), 64 hops max, 52 byte packets
1 128.119.232.1 (128.119.232.1)  5.001 ms  1.859 ms  1.879 ms
2 core1-rtxe-0-1-1.gw.umass.edu (128.119.0.237)  1.843 ms  1.929 ms  1.811 ms
3 border1-rte-5-0-0.gw.umass.edu (192.80.83.102)  1.817 ms  2.185 ms
   border2-rte-5-0-0.gw.umass.edu (192.80.83.110)  2.013 ms
4 5-2-19.ear3.newyork1.level3.net (4.71.230.233)  7.528 ms
   border2-rte-4-0-1.gw.umass.edu (192.80.83.97)  2.042 ms  2.105 ms
5 5-2-19.ear3.newyork1.level3.net (4.71.230.233)  7.413 ms
   ae-1-3110.edge4.frankfurt1.level3.net (4.69.163.106)  84.297 ms
5-2-19.ear3.newyork1.level3.net (4.71.230.233)  7.824 ms
6  84.442 ms
7  * te2-2.c102.f.de.plusline.net (212.162.24.58)  86.318 ms  87.387 ms
8  * * *
9 212.19.61.13 (212.19.61.13)  85.865 ms !X * *
10 * * 212.19.61.13 (212.19.61.13)  85.324 ms !X

Shortest Path Problems

- Represent network of routers as graph with weighted edges
- Problem to solve:
  - Find path with smallest total weight
- Similar to BFS but here we are concerned with the total weight of the path, rather than number of nodes in path
- If all weight are equal, problem is the same
Routing – Intro

• Shortest path routing
• Centralized approach
  • Each node has full “view” of network
  • Each node calculates shortest path using routing algorithm
  • “Link state algorithm”
  • (Exchange of link information always decentralized)

Routing - Intro

• Distributed approach
  • Each node computes best path without full view
  • Shortest path computed as link information is exchanged
  • “Distance vector algorithm”
**Link State Algorithm**

- Link cost of all links is broadcast to all nodes
- Dijkstra’s algorithm to find shortest path to all nodes
  - Each node calculates its own tree
- Notation:
  - \( D(v) \) is least cost to \( v \) in current iteration
  - \( p(v) \) is previous node along least cost path
  - \( N' \) is subset of nodes with guaranteed least cost paths

Algorithm:

Initialization:

\[ N' = \{u\} \]

For all nodes \( v \): if neighbor of \( u \) then \( D(v) = c(u,v) \),
else \( D(v) = \infty \)

Loop until \( N' = N \):

- Find \( w \in N' \) with minimum \( D(w) \) and add \( w \) to \( N' \)
- For each neighbor \( v \) of \( w \) (\( v \not\in N' \)):
  \[ D(v) = \min(D(v), D(w) + c(w,v)) \]
Link State Algorithm

```python
from Graph import Graph, Vertex, PriorityQueue

def dijkstra(aGraph, start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(), v) for v in aGraph])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() + currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
                nextVert.setDistance(newDist)
                nextVert.setPred(currentVert)
                pq.decreaseKey(nextVert, newDist)
```

PQ = x,v,w

![Graph Diagram]

Link State Algorithm
Link State Algorithm

ECE 241 – Data Structures Fall 2021 © 2021 Mike Zink

PQ = vy,w

ECE 241 – Data Structures Fall 2021 © 2021 Mike Zink

Link State Algorithm

ECE 241 – Data Structures Fall 2021 © 2021 Mike Zink

PQ = yw
Link State Algorithm

PQ = wz

Link State Algorithm

PQ = z
Link State Algorithm

PQ = None

<table>
<thead>
<tr>
<th>step</th>
<th>N’</th>
<th>D(b),p(b)</th>
<th>D(c),p(c)</th>
<th>D(d),p(d)</th>
<th>D(e),p(e)</th>
<th>D(f),p(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Link State Algorithm

- **Diagram**
  - Nodes: a, b, c, d, e, f
  - Edges: a-b (1), b-c (5), c-a (1), a-d (3), d-c (1), c-f (2), f-e (4), e-d (2), e-b (7)

- **Table**
<table>
<thead>
<tr>
<th>Step</th>
<th>N'</th>
<th>D(b),p(b)</th>
<th>D(c),p(c)</th>
<th>D(d),p(d)</th>
<th>D(e),p(e)</th>
<th>D(f),p(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{a}</td>
<td>1,a</td>
<td>∞</td>
<td>3,a</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>{a,b}</td>
<td>6,b</td>
<td>2,b</td>
<td>5,b</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{a,b,d}</td>
<td>6,b</td>
<td>4,d</td>
<td>9,d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{a,b,d,e}</td>
<td>5,e</td>
<td></td>
<td></td>
<td>8,e</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{a,b,c,d,e}</td>
<td></td>
<td></td>
<td></td>
<td>7,c</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{a,b,c,d,e,f}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Knights Tour Problem

- **Puzzle**
  - Played on chess board with single figure, the knight

- **Objective**
  - Find sequence of moves that allow knight to visit every square on board “exactly” once

- **Such sequence is called “tour”**

- **Upper bound on possible tours is 1.35 * 10^{35}**

- **Use graph search to solve problem**
Knights Tour Problem

Solve problem by using two main steps:

• Represent legal moves of knight on chessboard as graph
• Use a graph algorithm to find path of length \( \text{rows} \times \text{columns} - 1 \) where every vertex on graph is visited exactly once

• Each square represented as node in graph
• Each legal move represented by edge
Building the Graph

from Graph import Graph

def knightGraph(bdSize):
    ktGraph = Graph()
    for row in range(bdSize):
        for col in range(bdSize):
            nodeId = posToNodeId(row, col, bdSize)
            newPositions = genLegalMoves(row, col, bdSize)
            for e in newPositions:
                nid = posToNodeId(e[0], e[1], bdSize)
                ktGraph.addEdge(nodeId, nid)
    return ktGraph

def posToNodeId(row, column, board_size):
    return (row * board_size) + column

def genLegalMoves(x, y, bdSize):
    newMoves = []
    moveOffsets = [(-1, -2), (-1, 2), (1, -2), (1, 2), (-2, -1), (-2, 1)]
    for i in moveOffsets:
        newX = x + i[0]
        newY = y + i[1]
        if legalCoord(newX, bdSize) and legalCoord(newY, bdSize):
            newMoves.append(((newX, newY))
    return newMoves

def legalCoord(x, bdSize):
    if x >= 0 and x < bdSize:
        return True
    else:
        return False
**Complete Graph**

- 336 edges
- Less connections for vertices on edges of board
- Sparsity:
  - Fully connected graph: 4096 edges
  - Matrix only 8.2% filled

**Depth First Search (DFS)**

- Solve problem with depth first search (DFS) algorithm
- Creates search tree by exploring one branch of the tree as deeply as possible
- We will look at two algorithms:
  1. Directly solves problem by explicitly forbidding a node to be visited more than once
  2. More general, but allows nodes to be visited more than once as the tree is constructed
Implementing Knight’s Tour

- DFS exploration of graph finds path with exactly 63 edges
- When dead end is found (more moves possible)
  - Algorithm backs up tree to next deepest vertex allowing a legal move

```python
def knightTour(n, path, u, limit):
    u.setColor('gray')
    path.append(u)
    if n < limit:
        nbrList = list(u.getConnections())
        i = 0
        done = False
        while i < len(nbrList) and not done:
            if nbrList[i].getColor() == 'white':
                done = knightTour(n+1, path, nbrList[i], limit)
                i = i + 1
            else:
                done = True
    return done
```
DFS – Coloring

- DFS uses colors to keep track which vertices have been visited
  - White: unvisited
  - Gray: visited
- If neighbors of particular vertex have been explored && length of vertices < 64 => dead end reached
- If dead end reached => backtrack (Return from knightTour with false)

Since DFS is recursive, use stack to help with backtracking

- After return from knightTour with status False:
  - Remain inside while loop
  - Look at nextvertex in nbrlist
Simple Example

- Following figures show steps of search
- Assume `getConnections` orders nodes in alphabetical order
- Start with calling `knightTour(0,path,A,6)`

```
A
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>
```

- `knightTour` starts with node A (a))
- B and D are adjacent to A
- Since B comes next in alphabet, it is chosen next (b))
- Recursively calling `knightTour` explores B
Simple Example

- B is adjacent to C and D
- `knightTour` elects to explore C
- C is dead end with no adjacent white notes (c))
- Change color of C back to white (d))
- Backtracks search to vertex B

Simple Example

- Next vertex to explore is D (e))
- `knightTour` makes recursive calls until we get to node C again (f), (g), (h))
Simple Example

• when we get to node C the test \( n < \text{limit} \) fails
• \( \Rightarrow \) all nodes in graph exhausted
• return True to indicate that we have made a successful tour of the graph
• return the list, path has the values [A,B,D,E,F,C], which is the the order we need to traverse the graph to visit each node exactly once

Simple Example

• Complete tour around 8 x 8 board
Next Steps

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