GRAPHS, SHORTEST PATH

ANA BELL and FRÉDO DURAND

MIT EECS, 6.00
ONLINE SUBJECT EVALUATIONS ARE NOW OPEN

http://web.mit.edu/subjectevaluation

• You have until Monday, May 19 at 9 AM
• Please evaluate all subjects in your list
• Don’t forget your TAs
• Write comments

Your feedback is read and valued!
DECISION TREES

• A couple of lectures ago - knapsack problem
• Set of items a,b,c,d
• Each decision
  • left path (include item), right path (don’t include)
ROOTED BINARY TREE

- Top node
  - No parents
    - Root
- Non-root nodes
  - One parent
- Childless node
  - Leaf
- All nodes
  - At most two children
TREE SIZE

• Traversal
  • left first depth first

• Size of tree
  • `len(items)` depth
  • At most $2^{\text{level}}$ nodes at each level
NEW OPTIMIZATION PROBLEM

• Trees are a type of graph
• Graphs solve a more general problem
  • Not constrained to 2 children per node
  • Do not have one root node

• Ex. Prices for flights between two cities with data between many cities
  Find route with fewest stops
  Find least expensive airfare
  Find least expensive airfare with < two stops
GRAPH LINGO

• Set of objects
  • Nodes or vertices

• Objects are connected by
  • Edges or arcs
  • Edges can be uni or bidirectional or weighted

• Road maps, internet, protein interactions, movie similarities
UNDIRECTED GRAPH

• All nodes reachable from each other
  • Path from every node to every other node

• Ex. Road from city A ➔ city B
DIRECTED GRAPH aka DIGRAPH

• Unidirectional edges
  • From parent to child node

• Nodes may not be reachable from all other nodes
  • No path from A → C

• Ex. Have one way streets
• Ex. Links from one page to another, not necessarily vice versa
WEIGHTED DIRECTED GRAPH

• Extra information for each edge
  • Weight

• Shortest paths
  • Must explore all paths
  • Path C-B is now through D

• Ex. Add tolls to roads
• Ex. Internet traffic, weight can be how often someone clicks a link
class Node(object):
    def __init__(self, name):
        self.name = str(name)
    def getName(self):
        return self.name
    def __str__(self):
        return self.name

class Edge(object):
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def __str__(self):
        return str(self.src) + '->' + str(self.dest)

class WeightedEdge(Edge):
    def __init__(self, src, dest, weight = 1.0):
        self.src = src
        self.dest = dest
        self.weight = weight
    def getWeight(self):
        return self.weight
    def __str__(self):
        return str(self.src) + '->(' + str(self.weight) + ')\' + str(self.dest)
GRAPH REPRESENTATIONS

• Adjacency matrix (N nodes)
  • N x N matrix
  • Entries are weights or True/False
  • Careful with multiple edges between 2 nodes

• Adjacency list
  • List edges emanating from each node
ADJACENCY MATRIX

A | B | C | D
---|---|---|---
A | x |
B | x | x | x |
C | x | x |
D | x | x |

A | B | C | D
---|---|---|---
A |   |
B | x |
C | x | x |
D | x |

A | B | C | D
---|---|---|---
A |   |
B |   |
C | 7 | 2 |
D | 3 |   |

A

B

C

D

A

B

C

D

A

B

C

D

1

7

2

3
ADJACENCY LIST

A: B
B: A, C, D
C: B, D
D: B, C

A: A
B: B
C: B, D
D: B

A: (A,1)
B: (B,7), (D,2)
C: (B,3)
WHICH REPRESENTATION IS BETTER

• Depends on application

• Adjacency matrix
  • Better for dense graphs
  • Wastes space if only a few cells filled in

• Adjacency list
  • Better for sparse graphs
  • Arguably less intuitive to visualize
class Digraph(object):
    def __init__(self):
        self.nodes = []
        self.edges = {}

    def addNode(self, node):
        if node in self.nodes:
            raise ValueError('Duplicate node')
        else:
            self.nodes.append(node)
            self.edges[node] = []

    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not(src in self.nodes and dest in self.nodes):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)

    def childrenOf(self, node):
        return self.edges[node]

    def hasNode(self, node):
        return node in self.nodes

    def __str__(self):
        res = ''
        for k in self.edges:
            for d in self.edges[k]:
                res = res + str(k) + '->' + str(d) + '
        return res[:-1]

class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)
GRAPH OPTIMIZATION PROBLEMS

• Shortest Path

• Shortest Weighted Path

• Cliques

• Min Cut
SHORTEST PATH

•Source node A
•Destination node B
•Shortest sequence of edges $A \rightarrow B$ such that
  •Edges follow in sequence
  •Destination node of one edge is the source node of another edge
SHORTEST WEIGHTED PATH

• Define function on weights of edges and minimize
  • Function can be sum

• Not shortest sequence, but smallest total weight
  • At the cost of traversing a few more edges

• More common problem
  • Driving directions
  • Minimize time or distance (weights you care about)
CLIQUES

• Set of nodes such that there is a path between each pair in set

• Social clique
  • Group of people that can contact to each other
  • Guarantees every member of clique can reach any other member
MIN CUT

• Given a graph and 2 sets of nodes, find minimum number of edges to remove such that two sets are disconnected
SHORTEST PATH WITH DEPTH FIRST SEARCH

• Recursive

Start by choosing one child of node currently at
Keeps doing that until
  Reached node already seen (avoid cycles)
  Reached node need to get at
  Reached a node with no kids
If reached node with no children, backtrack and take next child of node it was at.
  → Systematically explore all possible paths
  → Along the way keep track of best one
```python
def DFS(graph, start, end, path = [], shortest = None):
    # assumes graph is a Digraph
    # assumes start and end are nodes in graph
    path = path + [start]
    print 'Current dfs path:', printPath(path)
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if node not in path: # avoid cycles
            if shortest == None or len(path)<len(shortest):
                newPath = DFS(graph, node, end, path, shortest)
                if newPath != None:
                    shortest = newPath
    return shortest
```
DFS EXAMPLE – SHORTEST PATH 0 → 5

- Start at source node
  - 0
- Children of 0?
  - 1
  - 2

Path traversed so far:

0

Length: 0
DFS EXAMPLE – SHORTEST PATH $0\rightarrow5$

• Pick a child of 0
  • Smallest of the two
  • 1

Path traversed so far: $0\rightarrow1$
Length: 1
DFS EXAMPLE – SHORTEST PATH 0→5

• Pick a child of 1
  • 0 or 2
  • Have visited 0
  • Pick 2

Path traversed so far: 0→1→2
Length: 2
DFS EXAMPLE – SHORTEST PATH 0→5

• Pick a child of 2
  • 3 or 4
  • Pick smallest, 3

Path traversed so far: 0→1→2→3
Length: 3
DFS EXAMPLE – SHORTEST PATH 0→5

• Pick a child of 3
  • 1 or 4 or 5
  • Have visited 1
  • 4

• Pick a child of 4
  • Only 0, already visited

Path traversed so far:
  0→1→2→3→4

Length: 4
DFS EXAMPLE – SHORTEST PATH 0→5

• Backtrack back one
  • Node 3’s other kids
    • Already been to 1
    • Just came from 4
    • Pick 5

Path traversed so far: 0→1→2→3→5
Length: 4
DFS EXAMPLE – SHORTEST PATH 0→5

• 5 has no kids
• Backtrack back one
  • Already visited all kids of 3
• Backtrack back one
  • Other kid of 2 is 4
• 4’s only kid is 0

Path traversed so far: 0→1→2→4
Length: 3
DFS EXAMPLE – SHORTEST PATH 0→5

• Backtrack back one
  • Visited all kids of 2
• Backtrack one more
  • Visited all kids of 1
• Backtrack one more
  • Pick 2

Path traversed so far:
  0→2
Length: 1
DFS EXAMPLE – SHORTEST PATH 0→5

• Kids of 2
  • 3 and 4
  • Pick 3

Path traversed so far: 0→2→3
Length: 2
DFS EXAMPLE — SHORTEST PATH 0→5

• Kids of 3
  • 1, 4, 5
  • Pick 1
• Kids of 1 we have seen

Path traversed so far:
  0→2→3→1
Length: 3
DFS EXAMPLE – SHORTEST PATH 0→5

• Backtrack to 3
  • Pick 4
• Kid of 4 we have seen

Path traversed so far:
  0→2→3→4
Length: 3
DFS EXAMPLE – SHORTEST PATH 0→5

• Backtrack to 3
  • Pick 5
• Reach our destination
  • Note length of path
  • Shorter than before

Path traversed so far: 0→2→3→5
Length: 3
DFS EXAMPLE – SHORTEST PATH 0→5

• Backtrack to 3
  • Visited all kid nodes
• Backtrack to 2
  • Pick 4
• Already visited kid of 4

Path traversed so far:
0→2→4
Length: 2
DFS EXAMPLE – SHORTEST PATH 0→5

• Backtrack to 2
  • Visited all kids
• Backtrack to 0
  • Visited all nodes
• Done
DFS SHORTEST PATH 0→5

- 0
- 0→1
- 0→1→2
- 0→1→2→3
- 0→1→2→3→4
- 0→1→2→3→5 ← length 4
- 0→1→2→4
- 0→2
- 0→2→3
- 0→2→3→1
- 0→2→3→4
- 0→2→3→5 ← length 3
- 0→2→4
THOUGHTS ON DFS

• To get global shortest path
  • Search entire tree for all possible paths
  • May not find global SP if stop early

• Weighted edges
  • Will not necessarily find shortest path
  • Can modify the algorithm
BREADTH FIRST SEARCH

• Start at source node, picks every child before moving down one level

Iterate through every child of start node
Keeps doing that until
  Reached node need to get at
  If destination node not found, start nodes become each children of start node
BFS CODE

def BFS(graph, start, end, pathQueue = []):
    # assumes graph is a Digraph
    # assumes start and end are nodes in graph
    initPath = [start]
    pathQueue.append(initPath)
    visited = [start] # to avoid going to nodes have already seen
    while len(pathQueue) != 0:
        # Get and remove oldest element in pathQueue
        tmpPath = pathQueue.pop(0)
        lastNode = tmpPath[len(tmpPath) - 1]
        print 'Current BFS path:', printPath(tmpPath)
        if lastNode == end:
            return tmpPath
        for linkNode in graph.childrenOf(lastNode):
            if linkNode not in visited:
                newPath = tmpPath + [linkNode]
                pathQueue.append(newPath)
                # uncomment this line to add node to nodes have seen
                # visited.append(linkNode)
                return None
BFS EXAMPLE – SHORTEST PATH 0→5

• Enumerate kids of start node
  • 1 and 2

Path traversed so far:
  0
Length: 0
BFS EXAMPLE – SHORTEST PATH 0\rightarrow 5

• Go through each kid
  • Is 1 the end node?
    • No
    • Keep going to other kids

Path traversed so far:
  \text{0} \rightarrow \text{1}

Length: 1
BFS EXAMPLE – SHORTEST PATH 0→5

• Go through each kid
  • Is 2 the end node?
    • No
• Node 0 has no other kids
  • Go through kids of node 1

Path traversed so far: 0→2
Length: 1
BFS EXAMPLE – SHORTEST PATH 0→5

- Node 1 has only one kid
- Is 2 the end node?
  - No
- Look at other nodes on same level as 1

Path traversed so far: 
0→1→2

Length: 2
BFS EXAMPLE – SHORTEST PATH 0→5

- Node 2 has 2 kids
  - 3 and 4
- Is 3 the end node?
  - No

Path traversed so far:
0→2→3
Length: 2
BFS EXAMPLE – SHORTEST PATH $0 \rightarrow 5$

- Node 2 has 2 kids
  - 3 and 4
- Is 4 the end node?
  - No
- Look at nodes on next level

Path traversed so far:

$0 \rightarrow 2 \rightarrow 4$

Length: 2
BFS EXAMPLE – SHORTEST PATH 0→5

- Through node 1, node 2 has 2 kids
  - 3 and 4
- Is 3 the end node?
  - No

Path traversed so far:
  0→1→2→3
Length: 3
BFS EXAMPLE – SHORTEST PATH 0→5

• Through node 1, node 2 has 2 kids
  • 3 and 4
• Is 4 the end node?
  • No
• Keep going with this level

Path traversed so far: 0→1→2→4
Length: 3
BFS EXAMPLE – SHORTEST PATH 0→5

• Node 3 has 3 kids
  • 1, 4, 5
• Is 1 the end node?
  • No

Path traversed so far:
  0→2→3→1

Length: 3
BFS EXAMPLE – SHORTEST PATH 0→5

• Node 3 has 3 kids
  • 1, 4, 5
• Is 4 the end node?
  • No

Path traversed so far:
  0→2→3→4

Length: 3
BFS EXAMPLE – SHORTEST PATH 0→5

- Node 3 has 3 kids
  - 1, 4, 5
- Is 5 the end node?
  - Yes, stop here
  - Going farther will no longer be shortest path

Path traversed so far:
0→2→3→5
Length: 3
BFS SHORTEST PATH 0 → 5

- 0
- 0 → 1
- 0 → 2
- 0 → 1 → 2
- 0 → 2 → 3
- 0 → 2 → 4
- 0 → 1 → 2 → 3
- 0 → 1 → 2 → 4
- 0 → 2 → 3 → 1
- 0 → 2 → 3 → 4
- 0 → 2 → 3 → 5

Running algo with adding nodes already seen to a visited list. If see them again later, then skip these paths (know they will be longer than what have already seen)

← length 3
THOUGHTS ON BFS

• Finds optimal solution
  • Does not need to enumerate all paths
  • Every node reached through shortest path
• Caveat !!
  • All edge weights must be the same
  • Only true for unweighted graphs

• Weighted graphs?
  • Can’t assume node at next level is nearest
  • Keep distance estimate for each node
SUMMARY

• Graphs
  • Nodes and edges
  • Unidirectional, bidirectional, weighted
  • Adjacency matrix vs. adjacency list representation

• DFS
  • Recursive
  • Search all possible paths to find shortest

• BFS
  • Finds shortest path without searching all paths, guaranteed only on unweighted graphs
YOU
GO TALK TO YOUR FRIENDS
TALK TO MY FRIENDS
TALK TO ME
WEIGHTED GRAPH
SHORTEST PATH
SHORTEST WEIGHTED PATH
CLIQUES
MIN CUT