Graph and Tree Traversals

COMPSCI 230 — Discrete Math

March 29, 2016
Overview

1. Tree Traversals Revisited

2. Traversing Connected Graphs
   - Building a Spanning Tree
   - Depth-First Traversal of a Connected Graph
   - Breadth-First Traversal of a Connected Graph
   - Proofs that it Works
Recall the Dictionary Tree
Vertex Info, Edge Weights

- In the spelling checker app, we need to keep track of “words” along a path
- Vertices had just a True/False Boolean flag
- The dictionary tree was a somewhat unusual tree
- Often, vertices store more information
- Often, edges also have weights that measure distance or cost
- Weight may be 1, so each edge has a unit cost
- With weights, we may want to keep track of overall distance or cost along a path, rather than “words”
Traversing versus Visiting a Vertex

- A graph (or tree) traversal algorithm has a “current vertex”
- *Traversing* a vertex means making the vertex current
- When forming a path (such as a tour of a graph), each vertex is added to the path when it is traversed
- *Visiting* a vertex means accessing the vertex’s information
- We may traverse a vertex without visiting it
- ... but we cannot visit a vertex without traversing it
- You may just go through the St. Louis airport
- ... or you may stop and visit the city
Tree Traversals Revisited: Depth-First

class vertex:
    def __init__(self, info=None, nc=0):
        self.info = info
        self.children = [None] * nc

def tdf(u, order, visit, weight, total):
    if u:
        if order == 'pre': visit(u, total)
        for v in u.children:
            tdf(v, order, visit, weight, total + weight(u,v))
        if order == 'post': visit(u, total)

• Traverses the current vertex \(u\) once when `tdf` is called and once for each non-null \(v\) every time `tdf` returns
• Does not do anything with the vertices it traverses
• If we want to visit \(u\) exactly once, we can do it before or after visiting its children
• We can also tell each vertex we visit how much it costs to reach it from the root
Some Reasonable Defaults

def tdf(u, order='pre',
    visit=lambda u, total: print(u.info, ', depth', total),
    weight=lambda u, v: 1, total=0):
    if u:
        if order == 'pre': visit(u, total)
        for v in u.children:
            tdf(v, order, visit, weight, total + weight(u, v))
        if order == 'post': visit(u, total)

However, the essentials are what matters:

def tdf(u, order='pre',
    visit=lambda u: print(u.info)):
    if u:
        if order == 'pre': visit(u)
        for v in u.children: tdf(v, order, visit)
        if order == 'post': visit(u)
Depth-First Example

0, depth 0
1, depth 1
2, depth 2
3, depth 2
4, depth 1
5, depth 2
6, depth 2
def tbf(root, visit=lambda u: print(u.info)):
    if root:
        q = []
        q.append(root)
        while len(q) > 0:
            u = q.pop(0)
            visit(u)
            for v in u.children:
                if v: q.append(v)
def tbf(root,
    visit=lambda u, total:
        print(u.info, ', depth', total),
    weight=lambda u, v: 1, total=0):
    if root:
        q = []
        q.append((root, 0))
        while len(q) > 0:
            (u, total) = q.pop(0)
            visit(u, total)
            for v in u.children:
                if v: q.append((v, total+weight(u, v)))
Breadth-First Example

0, depth 0
1, depth 1
4, depth 1
2, depth 2
3, depth 2
5, depth 2
6, depth 2
Traversing Connected Graphs

- To traverse a connected graph $G$, we look for a spanning tree $T$ in it:
  - $T$ is a subgraph of $G$ that is a tree
- Conceptually, we can first build $T$ and then traverse it with either tdf or tbf
- However, to build $T$ we also need to traverse it, so construction and traversal occur together
Building a Spanning Tree

- In a rooted tree, every vertex but the root has one parent, and every non-leaf vertex has children.
- In a connected graph, every vertex has neighbors: there is no distinction between parent and children.
- Basic idea for building a spanning tree $T$ for a connected graph $G$:
  - Pick any vertex $u$ of $G$ as the root of $T$. Add $u$ to $T$.
  - Think of each neighbor $v$ of $u$ in $G$ as a child of $u$ in $T$. Add $(u, v)$ and $v$ to $T$ except when $v$ is already in $T$.
  - Repeat for all children of $u$ in some order.
- The exception prevents loops and keeps $T$ a tree.
- Every vertex will eventually be added to $T$ because $G$ is connected: $T$ is a spanning tree.
- Different orders lead to different trees.
Depth-First Traversal of a Connected Graph

A possible implementation

[Assumes neighbors and print(u) to do the right thing.]

```python
def gdf(G, u, visit = lambda u: print(u)):
    def span(G, u):
        nonlocal VT
        # So all VT are the same
        visit(u)
        for v in neighbors(u, G):
            if v not in VT:
                VT.append(v)
                span(G, v)
        VT = [u]  # Keeps track of vertices traversed
        span(G, u)
```
If You Want the Tree
Also keep track of edges, not just vertices
[Assumes that graph operators are implemented.]

```python
def gdf(G, u):
    def span(G, u):
        nonlocal T
        for v in neighbors(u, G):
            if v not in V(T):
                # V(T): vertices in T
                addEdge((u, v), T)  # Also adds v
                span(G, v)
        T = emptyGraph()
        addVertex(u, T)
        span(G, u)
        return T

Stores T as a graph. No visit. We can visit T later
Depth-First Traversal Example
def gbf(G, u, visit=lambda u: print(u)):
    q = []
    VT = []
    q.append(u)
    while len(q) > 0:
        u = q.pop(0)
        visit(u)
        VT.append(u)
        for v in neighbors(u, G):
            if v not in VT and v not in q:
                q.append(v)
If You Want the Tree

def gbf(G, u, visit=lambda u: print(u)):
    q = []
    T = emptyGraph()
    q.append((u, None))
    while len(q) > 0:
        (u, parent) = q.pop(0)
        visit(u)
        addEdge((parent, u), T)  # Also adds u
        for v in neighbors(u, G):
            if v not in VT and v not in q:
                q.append((v, u))
    return T
Breadth-First Traversal Example
Problems

• Basic idea for building a spanning tree $T$ for a connected graph $G$:
  • Pick any vertex $u$ of $G$ as the root of $T$. Add $u$ to $T$
  • Think of each neighbor $v$ of $u$ in $G$ as a child of $u$ in $T$. Add $(u, v)$ and $v$ to $T$ except when $v$ is already in $T$
  • Repeat for all children of $u$ in some order

• The exception prevents loops and keeps $T$ a tree
• Every vertex will eventually be added to $T$ because $G$ is connected: $T$ is a spanning tree
• (Somewhat) more formally ...
• [Note: a loop is a closed path]
Proving that $T$ is a Connected Tree

- We add the root vertex to $T$ when $T$ is empty
- Once the root is in $T$, it will no longer be traversed
- Therefore, there is no loop back to the root in $T$
- If there are additional vertices in $T$, then $u$ will be connected to its neighbors
- For all other vertices $v$:
  - We add $v$ to $T$ iff $v \notin T$ and $(u, v) \in T$ for some $u$
  - So $v$ is connected to some other vertex in $T$
  - Before insertion, since $v$ is not in $T$ there are no edges incident to $v$ in $T$
  - So $u$ is the only parent of $v$
  - Since now $v \in T$, it will be the child of no other vertex in $T$
  - Therefore, there is no loop back to $v$ in $T$
- In $T$, there is no loop back to any vertex and every vertex is connected to some other vertex: $T$ is a connected tree
Proving that $T$ Spans $G$

- Proof by contradiction: Suppose that some vertex of $G$ is not in $T$ after the construction
- Divide all vertices in $G$ into sets $\{u \in T\}$ and $\{u \notin T\}$
- $\{u \in T\}$ is not empty because the root is in it
- $\{u \notin T\}$ is not empty because of our tentative assumption
- Since $G$ is connected, there is a path from every vertex in one set to every vertex in the other
- Since both sets are nonempty, there are nonempty paths
- There must be an edge $(u, v)$ that crosses the boundary between the two sets along a path
- Let $u \in T$ and $v \notin T$
- Since $u \in T$, it was traversed during construction of $T$
- Since $v \notin T$, it was added during construction of $T$, along with $(u, v)$
- Therefore $v \in T$, a contradiction
- All vertices of $G$ are in $T$, and therefore $T$ spans $G$