CS472 Module 2 Part D - Graph Traversal

Athens State University

2013-11-18 Mon

Outline

Review: Brute force and graphs

- Brute-force algorithms: A method of computation wherein all permutations of a problem are tried manually until one is found that provides a solution, in contrast to the implementation of a more intelligent algorithm.
- Graph: A pair G of sets (V, E) where V is the set of vertices of G and E is the set of edges connecting those vertices.
  - A graph can be directed or undirected.

Review: Data Structures: Adjacency Lists and Matrices

Comparison	Winner
Faster to test if edge in graph	Adj. matrix
Faster to find degree of matrix	Adj. list
Less memory on small graphs	Adj. List
Less memory on big graphs	Adj. matrix
Edge insertion or deletion	Adj. matrix
Faster to traverse the graph	Adj. list
Better for most problems	Adj. List

Graph Traversal: Motivation

- Most common operation on a graph is to systematically process each vertex and edge in the graph. This is called "graph traversal"
- You have already seen examples of traversal with in-order, pre-order, and post-order traversal of trees
- Almost all complex bookkeeping operations on graphs require application of graph traversal

Graph Traversal: Concept

- Think bread crumbs in a maze
  - Need to mark each vertex as traverse
  - Need to keep track of what has not been completely explored
- Each vertex in G will be assigned one of three states
  - Undiscovered: Vertex has not been visited
  - Discovered: Vertex has been found, but we have not yet checked out all of its incident edges
  - Processed: All incident edges have been visited
- Will need to maintain a list with vertices discovered but not yet processed

Graph Traversal: Concept

- At start, only our starting node will be discovered
- Consider each edge leaving a node v
  - If an edge (v, x) leads to a node x not yet discovered, mark it so and stick into our list
  - Can ignore edges going to a processed vertex as following that path will not provide new information
  - Can ignore edge going to a discovered vertex as that vertex is already on the list for processing

Graph Traversal: Concept

- Directed vs. undirected edges:
  - directed edges will be processed only once (from which node?)
  - undirected edges will be processed twice
- Every connected edge and vertex will eventually be visited
  - Suppose there is a vertex u that is undiscovered but whose neighbor v is discovered
  - Neighbor v will eventually get processed, upon which we will visit u

Graph Traversal: Breadth-First Search (BFS)

Algorithm 1: Breath-First Graph Traversal

```
Input: A graph G and staring vertex x
Output: A tree T containing the vertices of G
for each u ∈ V − s do
    state[u] ← "undiscovered";
    p[u] ← NULL ;
p[s] ← NULL;
Q = s;
while Q is not empty do
    u ← Dequeue (Q);
    Process (u);
    for v ∈ adj[u] do
        Process (edge(u,v));
        if state[v] = "undiscovered" then
            state[v] ← "discovered";
            p[v] ← u;
            Enqueue(Q,v);
    state[u] ← "processed";
```

Graph Traversal: BFS : Behavior

- BFS works in a "concentric" fashion
  - Start with a vertex and visit all adjacent nodes
  - Then look at all vertices two edges away, and so on
- Note the use of a queue to store the structure
- Builds a *breadth-first search forest*
  - Start is the root of first tree
  - *Tree edge*: discovered vertex attached as child to vertex from which it had been reached
  - *Cross edge*: edge leading to processed vertex other than its immediate predecessor

Graph Traversal: BFS: Analysis

- BFS is $O(|V|^2)$ for adjacency matrix representation
- BFS is $O(|V| + |E|)$ for adjacency lists

Graph Traversal: Depth-First Search (DFS)

Algorithm 2: Depth-First Graph Traversal

```
state[u] ← "discovered";
entry[u] ← time;
increment time;
Process (u);
for v ∈ Adj[u] do
    Process (Edge(u,v));
    if state[v] = "undiscovered" then
        p[v] ← u;
        dfs (G,v);
state[u] ← "processed";
exit[u] ← time;
increment time;
```

Graph Traversal: DFS: Implication of recursion

- What data structure is being used to store vertices that have been discovered but not processed?
  - *Queue*: By BFS using a queue, we explore oldest unexplored vertices first. Thus, we radiate out from our starting node.
  - *Stack*: By DFS using a stack (why?), we explore the vertices by following a path until we are surrounded by previously discovered vertices.
- Contrast BFS and DFS against in-order, pre-order, and post-order traversals of trees.

Graph Traversal: DFS: Use of time intervals

- *Who is an ancestor?*: Suppose x is an ancestor of y
  - This implies that we must enter x before y and exit y before x
  - Thus, we have nesting of time intervals within children.
- *How many descendants?*: The difference between exit and entry times for v tells us how many descendants v has in the DFS tree.
  - Clock is incremented on each entry and exit, so half the time difference denotes the number of descendants of v

Graph Traversal: DFS: Tree edge and back edges

- Tree edge: tree edges discover new vertices, and are those encoded in the parent relation in search tree
- Back edge: back edges are those whose other endpoint is an ancestor of the vertex other than the node being expanded
- Note that this gives us two very useful orderings of the vertices
  - The order vertices get pushed onto the stack gives us the order in which we see the vertices get reached for the first time
  - The order vertices get popped off the stack are the order in which vertices become dead end

Applications: Finding Paths in Unweighted Graphs

- Define the vertex that discovered vertex v as the parent of i
- In BFS, every vertex is discovered during the course of traversal which generates a tree of discovery with the initial search node as the root of the tree
- Vertices are discovered in order of increasing distance from the root
  - As such, the unique discovery tree path from the root to each node $x \in V$ uses the smallest number of edges possible on any path from the root to x

Applications: Finding Paths in Unweighted Graphs

So, if we want to find the path to get from a starting node to any node x, then assuming we have done a BFS, we can use the discovery tree to find the path.

- We have to work backwards from the endpoint to the start
- Have to have some way to reverse the path (either by using a stack or a recursion)

Applications: Connected Components: The six degrees of Kevin Bacon

- The theory is that in social networking graphs, one can always connect a person to Kevin Bacon, the actor, in six steps or less
- If true, then your social networking graph is an example of a connected graph
- The connected components of an undirected graph is the maximal pair of vertices such that there is a path between every pair of vertices

Applications: Connected Components: Algorithm

Algorithm 3: Connected Component Algorithm

Input: A graph G

Output: The set of connected components

```
component ← 0;
for i ∈ G do
    if i is not discovered then
        Add i to component;
        BFS (G,i);
```

Applications: Finding cycles

Back edges in a DFS tree are key to finding a cycle in an undirected graph

- If there are no back edges in the search tree, all edges are tree edges and no cycle exists in a tree
- But any back edge from a node to an ancestor creates a cycle

Very easy to determine this by making a simple enhancement to the edge processing in DFS

Applications: Finding cycles: Algorithm

Algorithm 4: Edge Processing DFS Modification for Finding Cycles

if *parent[x] is not y* **then**

```
    FindPath (y,x,parent);
    return path found and exit;
```

Key Points

- What is a graph?
- What is a graph traversal?
  - Sketch the design of the BFS and DFS algorithms
  - Performance implications of each algorithm
- Applications of graph traversal