CSE 326: Data Structures

Graphs

James Fogarty

Spring 2009
Application: Topological Sort

Given a graph, \( G = (V,E) \), output all the vertices in \( V \) sorted so that no vertex is output before any other vertex with an edge to it.

What kind of input graph is allowed? **DAG**

Is the output unique? No, often called a partial ordering
Topological Sort: Take One

1. Label each vertex with its *in-degree* (# of inbound edges)

2. **While** there are vertices remaining:
   a. Choose a vertex $v$ of *in-degree zero*; output $v$
   b. Reduce the in-degree of all vertices adjacent to $v$
   c. Remove $v$ from the list of vertices
void Graph::topsort(){
    Vertex v, w;

    labelEachVertexWithItsInDegree();

    for (int ctr=0; ctr < NUM_VERTICES; ctr++){
        v = findNewVertexOfDegreeZero();
        v.topologicalNum = counter;
        for each w adjacent to v
            w.indegree--;
    }
}

What's the bottleneck?
Finding a new vertex

Runtime: $O(|V|^2 + |E|) \rightarrow O(|V|^2)$
Topological Sort: Take Two

1. Label each vertex with its in-degree
2. Initialize a queue Q to contain all in-degree zero vertices
3. While Q not empty
   a. \( v = Q\text{.}\text{dequeue} \); output \( v \)
   b. Reduce the in-degree of all vertices adjacent to \( v \)
   c. If new in-degree of any such vertex \( u \) is zero
      \( Q\text{.}\text{enqueue}(u) \)

**Runtime:**
\[ O(|V| + |E|) \]

Is the use of a queue here important?

No, can use a stack, list, set, box, etc.

Changes behavior, but not the fact the result is a topological sort
```cpp
void Graph::topsort()
{
    Queue q(NUM_VERTICES);
    int counter = 0;
    Vertex v, w;
    labelEachVertexWithItsIn-degree();

    q.makeEmpty();
    for each vertex v
        if (v.indegree == 0)
            q.enqueue(v);

    while (!q.isEmpty()){
        v = q.dequeue();
        v.topologicalNum = ++counter;
        for each w adjacent to v
            if (--w.indegree == 0)
                q.enqueue(w);
    }
}
Topological Sort

• Applications:
  – Figuring out how to finish your degree
  – Computing the order in which to recompute highly interrelated spreadsheet shells
  – Determining the order to perform compilation tasks when executing a makefile
Graph Traversals

• Breadth-first and depth-first search work for arbitrary (directed or undirected) graphs
  – Require marking visited vertices.
  – Why?
  – So you do not go into an infinite loop! It’s not a tree.

• Either can be used to determine connectivity:
  – Is there a path between two given vertices?
  – Is the graph (weakly/strongly) connected?

• Which one:
  – Uses a queue?
  – Uses a stack?
  – Always finds the shortest path (for unweighted graphs)?
The Shortest Path Problem

• Given a graph $G$, edge costs $c_{i,j}$, and vertices $s$ and $t$ in $G$, find the shortest path from $s$ to $t$.

• For a path $p = v_0 v_1 v_2 \ldots v_k$
  
  – *unweighted length* of path $p = k$
    (a.k.a. *length*)
  
  – *weighted length* of path $p = \sum_{i=0..k-1} c_{i,i+1}$
    (a.k.a *cost*)

• Let’s do unweighted first…
Depth-First Graph Search

Open – Stack

DFS( Start, Goal_test)
push(Start, Open);

repeat
  if (empty(Open)) then return fail;
  Node := pop(Open);
  if (Goal_test(Node)) then return Node;
  for each Child of node do
    if (Child not already visited) then push(Child, Open);
  Mark Node as visited;
end
Breadth-First Graph Search

Open – Queue

BFS( Start, Goal_test)
    enqueue(Start, Open);

repeat
    if (empty(Open)) then return fail;
    Node := dequeue(Open);
    if (Goal_test(Node)) then return Node;
    for each Child of node do
        if (Child not already visited) then enqueue(Child, Open);
    Mark Node as visited;
end
Comparison: DFS versus BFS

• Breadth-first search
  – Always finds shortest paths – optimal solutions
  – Marking visited nodes can improve efficiency, but even without this search guaranteed to terminate

• Depth-first search
  – Does not always find shortest paths
  – Must be careful to mark visited vertices, or you could go into an infinite loop if there is a cycle

• Is BFS always preferable?
DFS Space Requirements

• Assume:
  – Longest path in graph is length $d$
  – Highest number of out-edges is $k$

• DFS stack grows at most to size $dk$
  
  $d$ – 1 nodes visited,
  $k$ – 1 choices remaining at each

• For $k=10$, $d=15$, size is 150
BFS Space Requirements

• Assume
  – Distance from start to a goal is $d$
  – Highest number of out edges is $k$

• BFS Queue could grow to size $k^d$
  – Imagine a $k$-nary tree of height $d$

• For $k=10$, $d=15$, size is $1,000,000,000,000,000,000,000$
Idea

• For large graphs, DFS is hugely more memory efficient, *if we can limit the maximum path length to some fixed d*.

  – If we *knew* the distance from the start to the goal in advance, we could simply *not add any children to stack after level d*.

  – But what if we don’t know *d* in advance?
Iterative-Deepening DFS (I)

Open – Stack

Bounded_DFS(Start, Goal_test, Limit)
    Start.dist = 0;
    push(Start, Open);

    repeat
        if (empty(Open)) then return fail;
        Node := pop(Open);
        if (Goal_test(Node)) then return Node;
        if (Node.dist < Limit) then
            for each Child of node do
                if (Child not already i-visited) then
                    Child.dist := Node.dist + 1;
                    push(Child, Open);
                end
            end
        Mark Node as i-visited;
    end
Iterative-Deepening DFS (II)

IDFS_Search(Start, Goal_test)
    i := 1;
    repeat
        answer := Bounded_DFS(Start, Goal_test, i);
        if (answer != fail) then return answer;
        i := i+1;
    end
Analysis of IDFS

• Work performed with limit less than actual distance to G is wasted

• But the wasted work is usually small compared to amount of work done during the final iteration

$$\sum_{i=1}^{d} k^i = O(k^d)$$

Ignore low order terms!

Same time complexity as BFS

Same space complexity as (bounded) DFS
Saving the Path

• Our pseudocode returns the goal node found, but not the path to it

• How can we remember the path?
  – Add a field to each node, that points to the previous node along the path
  – Follow pointers from goal back to start to recover path
Example (Unweighted Graph)
Example (Unweighted Graph)
General Graph Search Algorithm

Open – some data structure (e.g., stack, queue)

Criteria – some method for removing an element from Open

Search( Start, Goal_test, Criteria)

insert(Start, Open);

repeat

if (empty(Open)) then return fail;

select Node from Open using Criteria;

if (Goal_test(Node)) then return Node;

for each Child of node do

if (Child not already visited) then

Child.previous := Node // if path desired

Insert( Child, Open );

Mark Node as visited;

end
**Single Source Shortest Paths (SSSP)**

- Given a graph $G$, edge costs $c_{i,j}$, and vertex $s$, find the shortest paths from $s$ to all vertices in $G$.

- Is finding paths to all the vertices harder or easier than the previous problem?
  - The same difficulty
    (imagine the one we want is the last one we reach)

- But we still haven’t dealt with edge costs…
Weighted SSSP: The Quest For Food

Can we calculate shortest distance to all nodes from Allen Center?
Weighted SSSP: The Quest For Food

Can we calculate shortest distance to all nodes from Allen Center?
Edsger Wybe Dijkstra (1930-2002)

• Invented concepts of structured programming, synchronization, weakest precondition, and "semaphores" for controlling computer processes. The Oxford English Dictionary cites his use of the words "vector" and "stack" in a computing context.

• Believed programming should be taught without computers

• 1972 Turing Award

• “In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.”
Shortest Path for Weighted Graphs

• Given a graph $G = (V, E)$ with edge costs $c(e)$, and a vertex $s \in V$, find the shortest (lowest cost) path from $s$ to every vertex in $V$

• Assume: only positive edge costs
Dijkstra’s Algorithm for Single Source Shortest Path

• Similar to breadth-first search

• But uses a heap instead of a queue:
  – Always select (expand) the vertex that has a lowest-cost path to the start vertex

• Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges