Graphs concluded

Michael P. Fourman

February 2, 2010

Paths again

In many graph applications, a fundamental problem is to determine whether a pair of vertices are connected via a sequence of zero or more edges, i.e., whether there is a *path* between the vertices. For example, if a logic circuit is being modelled by a graph, with vertices representing gates and edges representing wires, we might want to know whether a particular circuit input is connected to a particular circuit output.

In some applications,edges have associated weights, and we are interested in finding paths that are short (or long) in terms of the total weights of the edges on the path. For example, if a road map is being modelled by a graph, with vertices representing towns and weighted edges representing roads with distances, we might want to find the shortest route between two towns.

We have seen, in Lecture Note 13, how to use graph search to find which vertices are reachable from a given vertex, and how to find the shortest path between two vertices when each edge has length 1. We can find shortest paths in a weighted graph (with non-negative weights) using the same algorithm, placing the paths in a priority queue that gives shorter paths higher priority. This algorithm can also be adapted to find the shortest paths from a given vertex to all other vertices; when we first visit a vertex, we record the path that got us there. This variant is known as *Dijkstra's algorithm.*

The problem of finding *all* shortest paths, between every pair of vertices can be solved differently, using an algorithm due to Floyd and Warshall. We give an outline of the algorithm in English.

For a graph $G = (V, E)$, we consider the vertices in some order, $V = \{v_1, \ldots, v_n\}$, the algorithm constructs a sequence of graphs G_i for $0 \leq i \leq n$. All of the graphs have vertex set V. An edge joining (u, v) in graph G_i is a shortest path in G, linking u to v, *with any intermediate vertices drawn only from* $\{v_1, \ldots, v_i\}$. The graph G_0 is just G itself. An edge joining (s, t) in G_n is a shortest path between the vertices s and t in G. The subtlety of the

algorithm lies in the fact that it augments the paths by allowing only one new vertex to be involved at each stage.

Finding longest paths

Finding longest weight paths is important for critical path analysis in scheduling problems. However, the graphs must be acyclic, otherwise paths can have arbitrarily large weights. We can find longest paths just by negating all of the edge weights, and then using a shortest path algorithm. Unfortunately, Dijkstra's algorithm does not work if edges are allowed to have negative weights. However, the Floyd-Warshall algorithm does work, as long as there are no negative weight cycles, and so it can be used to find longest weight paths in acyclic graphs.

Topological sorting of a graph

In some applications involving the use of directed acyclic graphs, a requirement is to list the vertices of the graph in a linear ordering so that all vertices reachable from a given vertex come later than it in the list. An example is *scheduling,* where vertices represent activities and directed edges represent the fact that one activity must be performed before another: here the list would give an acceptable order in which to perform the activities. Constructing such a list is known as *topological sorting* of a directed graph. Note that it is not possible to topologically sort a directed graph that has cycles.

We can topologically sort a DAG using a variant of depth-first search. To sort all the vertices reachable from `x`, we must first sort all the vertices reachable from nodes adjacent to `x`, and then place x before this list. There are two complications: first, vertices may be reached by more than one path, the list `sorted` (which is itself topologically sorted) acts as a visited list; second, the graph may have cycles, the list `path` records the path by which the algorithm arrived at the current node, and is used to check for cycles.

On any call, `sort todo path sorted`, the list `sorted` is a topologically sorted list of vertices, closed under reachability; and if `c` is a member of `todo` then `x::path` is a (reversed) path in the graph. If the graph is acyclic, the call returns a topological sort of all vertices reachable from the `todo` list, or in the `sorted` list.

```
functor TOPSORT( structure G: GraphSig )=
struct
    type Graph = G.Graph
    type Vertex = G.Vertex

    exception Cycle

local
    fun member [] _ = false
      | member (h :: t) x = x = h orelse member t x
in
    fun topSort g =
        let fun sort [] path sorted = sorted
              | sort (x :: xs) path sorted =
                if member path x
                     then raise Cycle
                else if member sorted x
                     then sort xs path sorted
                else let val afterx = sort
                                        (G.adj g x)
                                        (x :: path)
                                         sorted
                     in
                         sort xs path (x :: afterx)
                     end
        in
            sort (G.vertices g) [] []
        end
end
end ;
```