1 Digraphs

A graph is a set of vertices $V$ and edges $E$. Vertices can be any class of objects with an equality operator. Edges are unordered or ordered pairs of vertices, in which case the graph is called undirected or directed respectively. We shall take the word 'graph' to mean either version and the word 'digraph' to refer to the directed version.

To make things clearer, we will deal with labelled digraphs, i.e. each vertex has a label. No two vertices in the same graph have the same label, so we can use labels to identify vertices. Just as important however, the labels have an independent existence outside the graph. A class representing a digraph must represent, at the very least, operations for

- creating and destroying a graph.
- adding, removing and checking the existence of, a vertex given a label. Note - removing a vertex also means removing any edges associated with it.
- adding, removing and checking the existence of, an edge given the labels of the vertices defining it.
- finding the number of edges and vertices in the graph.

A digraph can have both edges $(x, y)$ and $(y, x)$, but cannot have more than two edges $(x, y)$. Similarly, an undirected graph cannot have two edges involving the same pair of vertices. However, this restriction is lifted in some applications, in which case the edges are called parallel edges. We will allow $x$ to equal $y$, when we say there is a loop at $x$. ($x$ will thus be a neighbor of itself.)

If $(x, y)$ is an edge in a digraph, we say $x$ is adjacent to $y$ and $y$ is adjacent from $x$. Alternatively, $y$ is an out-neighbor of $x$, $x$ is an in-neighbor of $y$. $N_{in}(x)$ denotes the set of all in-neighbors of $x$, $N_{out}$ similarly defined. A vertex $x$ is a source iff there is no vertex $y$ such that $(y, x)$ is an edge, and a sink if there is no edge $(x, y)$.

The cost of graph algorithms is usually represented in terms of the number $|V|, |E|$ of vertices and edges respectively. Internally graphs can be represented using an adjacency matrix or an adjacency list. The first costs $\Theta(|V|^2)$ in space and the second $\Theta(|V| + |E|)$ in space. Many applications estimate that $|E|$ is nearer to $\Theta(|V|)$ than $\Theta(|V|^2)$ and thus use the latter.

Graph algorithms are often based on whether nodes are accessed depth-first or breadth-first.
2 Traversal

Traversal<STRUC> (D,v)
\% input : digraph D, vertex v of D
\% output : nothing
\% effect : visits each vertex of D accessible from v
mark v
STRUC.add (v)
while not STRUC.empty
    x = STRUC.remove ()
    N = {all unmarked vertices y adjacent from x}
    for each y in N
        mark (y)
        STRUC.add (y)

STRUC is some data structure that can you can add elements to and, more limingly, remove elements from in a predefined way that does not require an input argument. For example, it could be a stack or queue. \texttt{Traversal<stack>} is the Depth-First version, \texttt{Traversal<queue>} the Breadth-First version.

Here, a vertex said to be ‘marked’ when in it is placed in STRUC and ‘visited’ when it is removed. With a queue, the order is the same in both cases. If you want to do something to each vertex, e.g. print its contents, you do so when you visit it.

If STRUC is a stack, the Traversal algorithm can be written in recursive form easily as below. This time vertices are visited at the start of \texttt{Traversal\_rec}.

Traversal<STRUC> (D,v)
    mark v
    Traversal\_rec<STRUC> (D,v)

Traversal\_rec<STRUC> (D,x)
    N = {all unmarked vertices y adjacent from x}
    for each y in N
        mark y
        Traversal\_rec<STRUC> (D,y)

These algorithms may actually not visit all vertices in the digraph. They visit all vertices that can be got to from the initial vertex.
3 Searching

Suppose we want to know if there is a path from vertices v to w in a digraph. We could modify the above algorithms so that we get a list of all vertices visited in a call of Traversal(v) and then see if w is in the list. But this is overkill, and the modified version below is more appropriate.

ExistPath<STRUC> (D,v,w)
% input : digraph D, vertices v,w of D
% output : boolean value saying if there v→w path in D
mark v
STRUC.add (v)
while not STRUC.empty
  x = STRUC.remove ()
  if x equals w
    return TRUE
  N = {all unmarked vertices y adjacent from x}
  for each y in N
    mark (y)
    STRUC.add (y)
return FALSE

While it is good to know that there is a path from v to w, we might also want to know what such a path is. The naive thing to do is to, when we mark a vertex y, to also store a path from v to y. We can do so by storing this in a table Paths indexed by vertices. Note that if the path from v to y is v= v₀,...,vₖ, we could have chosen vₖ to equal y but expressly do not as that would be redundant.

FindPath<STRUC> (v,w)
% input : digraph D, vertices v,w of D
% output : some v→w path in D, or FALSE if no path exists.
mark v
Paths (v) = empty list
STRUC.add (v)
while not STRUC.empty
  x = STRUC.remove ()
  if x equals w
    return Paths (x)
  N = {all unmarked vertices y adjacent from x}
  for each y in N
    mark (y)
    Paths (y) = Paths (x) + x
    STRUC.add (y)
return FALSE

However, this is far more wasteful of space than it needs to be. Instead we can make use of the recursive relation Paths(y) = Paths(x) + x. Suppose we have a table PrevVtx that stores not the whole path, but just the last node of each path.
FindPath<STRUC> (v,w)
    mark v
    PrevVtx (v) = NULL
    STRUC.add (v)
    while not STRUC.empty
        x = STRUC.remove ()
        if x equals w
            return GetPaths (PrevVtx,v,x)
        N = {all unmarked vertices y adjacent from x}
        for each y in N
            mark (y)
            PrevVtx (y) = x
            STRUC.add (y)
        return FALSE

GetPaths(PrevVtx,v,x)
    if x == v
        return x
    else
        return GetPaths(PrevVtx,v,PrevVtx(x)) + x

This backtracking technique is useful in other situations.

4 Shortest Path problems

Suppose now that the edges have weights associated with them. For convenience, we will assume that all the weights are non-negative, but the condition we really need is weaker, namely that there be no negative-weight cycles in the digraph. This time we also have a table Cost such that at any time in the algorithm Cost(y) has the cost of the best path found so far from v to y. The length of a path is the sum of the weights of its edges. This time, STRUC has to be a queue.
FindShortestPath (D,v,w)
% Input: digraph D with weighted edges, vertices v,w of D
% Output: list representing shortest path from v to w in D, or
%       FALSE if no path exists.
for each vertex x
  Cost(x) = infinity
mark v
PrevVtx (v) = NULL
Cost (v) = 0
QUEUE.enqueue (v)
while not QUEUE.empty
  x = QUEUE.dequeue ()
  if x equals w
    return GetPaths (v,x) and Cost (x)
  N = {all unmarked vertices y adjacent from x}
  for each y in N
    NewCosty = Cost(x) + Cost(x->y)
    if NewCosty < Cost (y)
      Cost (y) = NewCosty
      mark (y)
      PrevVtx (y) = x
      QUEUE.enqueue (y)
  return FALSE

FindShortestPaths (v)
% Input: digraph D with weighted edges, vertex v of D
% Output: table Cost where Cost (x) has cost of shortest path from v to x,
%       and equals infinity if no path exists. Also returns
%       table PrevVtx from which shortest path can be found using GetPaths.
for each vertex x
  Cost(x) = infinity
mark v
PrevVtx (v) = NULL
Cost (v) = 0
QUEUE.enqueue (v)
while not QUEUE.empty
  x = QUEUE.dequeue ()
  N = {all unmarked vertices y adjacent from x}
  for each y in N
    NewCosty = Cost(x) + Cost(x->y)
    if NewCosty < Cost (y)
      Cost (y) = NewCosty
      mark (y)
      PrevVtx (y) = x
      QUEUE.enqueue (y)
  return Costs, PrevVtx

5 DAGs and Topological Sort

A special type of digraph one with no cycles. Its underlying graph (i.e. the graph you get by ignoring
directions on each directed edge) may have cycles, e.g. the graph with vertices 1,2,3 and edges 1 → 2, 2 → 3
and 1 → 3. Such a directed acyclic graph is generally known as a DAG.

A DAG represents a partial ordering in the following sense. Define a relation R on the vertices of a DAG by xRy iff there is a path in the DAG from x to y. R is reflexive since there is an (empty) path from any vertex to itself, antisymmetric because xRy AND yRx means there is a directed cycle (not necessarily including x or y) and transitive — so R is a partial order.

Theorem: any DAG E has a sink.

Proof: If not, then for any vertex v of E there is at least one vertex adjacent from it, call it s(v). If we start at any vertex, and keep going as defined by the algorithm below, we would go forever. A list of vertices visited would be infinite, and since the number of vertices is finite, some vertex will have to appear twice — which can only occur if the graph has a cycle.

Visit (v)
    Visit (succ(v))

Corollary: any DAG has a source.

Proof: reverse the edges of the DAG, apply the previous theorem to find a sink v, reverse edge direction again and now v is a source in the original DAG.

FindSink (D)
% input : non-empty DAG D
% output : some sink v of D
    pick any vertex v of D
    return FindSink_rec (D,v)

FindSink_rec (D,v)
    if there is a vertex w adjacent from v
        return FindSink_rec (D,w)
    else
        return v

A linear ordering v₁,...,vₙ of the vertices is said to be consistent with the partial ordering represented by the DAG iff vᵢRvⱼ ⇒ i ≤ j. Several linear orderings may be consistent with a partial ordering — the Topological Sort algorithm finds one of them.

Topological Sort (D)
% Input : DAG D
% Output : list representing a linear order consistent with D
% Effect : destructive algorithm - destroys D
    if D empty
        return NULL
    else
        v = FindSink (D)
        remove v from D
        return v + TopologicalSort(D)