Graph Algorithms

Graph algorithms are pervasive in Computer Science. The graph is such a general data structure that almost all computational problems can be formulated using one of the primary graph processing algorithms. Lists and trees are subsets of graphs. The major problems in hardware synthesis, operating system scheduling and sharing, compiler optimization, software design and minimization, network communication and synchronization, and requirements and specification modeling are graph problems. Variations of the fundamental graph algorithms of traversal, covering, and coloring model the problems in these and other application areas.

Types of graphs

A ***graph*** is a collection of *vertices* `V` (also called nodes) and *edges* `E` (also called arcs or links) which connect the vertices. A single edge can be described by the two vertices which it connects: `e = (u,v)`

undirected graph: The edges can be traversed in either direction. Undirected graphs are either trees or cyclic.

cyclic graph: The graph contains loops than permit a node to be visited recurrently.

directed graph: The edges are directional and support only one direction of traversal.

directed acyclic graph: There are no paths which permit visiting a node more than once.

weighted graph: The edges have a numerical value which may represent the cost of traversing that edge. Weights can be both positive and negative.

connected graph: All of the vertices can be visited when starting from any vertex. The list of vertices visited between a starting vertex and an ending vertex is called a *path*.

sparse graph: There are many more vertices than edges, that is `E << V^2`.

dense graph: The number of edges is about the same as the number of vertices squared; `E =~= V^2`

complete graph: Every pair of vertices has an edge connecting them. The graph is thus maximally connected.

clique: A subgraph (that is a subset of nodes and edges in a graph) which is complete.

bipartite graph: A graph with two sets of vertices, such that each edge connects vertices from different sets. No edges connect vertices in the same set.

hypergraph: A graph with edges that connect more than two vertices. All hypergraphs can be converted into regular bipartite graphs.

graph complement: The complement of a graph is a graph with the same vertices, but all edges exchanged (i.e. if an edge is present, then delete it; if an edge is missing, add it).

Representations of graphs

The choice of graph data structures depends on the density of the graph connectivity. The complexity of different algorithms depends both on the number of edges and on the number of vertices. Unlike sets, graphs often have a direct representation in memory. When the data field of a memory cell contains an address of another cell, the cell can be interpreted as a node and the shared address can be interpreted as a link.

adjacency lists

A collection of V lists (usually stored as an array), each listing the adjacent vertices of a particular vertex. For directed graphs, the sum of entries in the V lists will equal the number of edges. For undirected graphs, it will equal twice the number of edges. Adjacency lists use memory O(V+E). Use this for sparse graphs.

adjacency matrix

A square matrix of V Boolean entries, with a 1 recording adjacent vertices, and a 0 recording non-adjacent vertices. Use this for dense graphs.

indexed arrays

For graphs which change during processing, two arrays are used. The first array contains the V vertices, and provides an index for each vertex. Each indexed vertex in the vertex-array contains all the information about the vertex, including its processing status (visited, frontier, or not-visited). This array is changed only when the status of the vertex changes, the vertex index is never changed. The second array contains the E edges, and provides an index for each edge. The data field of the second array contains the indices of the two vertices connected by that edge. The connectivity of the graph is changed by changing the edge array entries. Graph traversal is managed by a priority queue of edge indices to be visited.

Topological Sort

Sorting lists and trees makes them easier to process, since the vertices have a consistent ordering. Directed acyclic graphs (DAGs) can also be sorted by the partial ordering implicit in the graph connectivity. Cyclic and undirected graphs do not have a consistent sort.

A ***topological sort*** of a DAG is a consistent ordering of the vertices that serves as an ordering for processing vertices. One convenient sorting is by length of shortest paths from the root vertex to the leaf vertices. The complexity of a topological sort is linear, O(V+E).

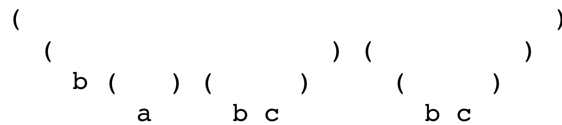
Interestingly, graph sorting is faster than both list and tree sorting, when the tree sort is done by comparing elements. There are linear-time list sorting algorithms (counting, radix and bucket sort). These rely on *a priori* encoding techniques for the input. Graph sort also relies on a particular type of input (i.e., an adjacency list). This structure is not, however, related to the type of input data, since data itself is sorted at each graph node.

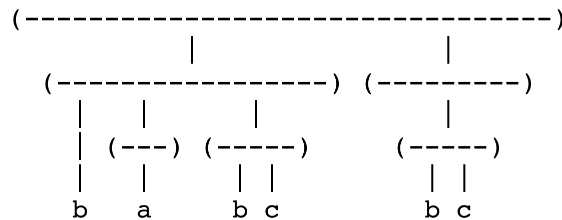
Parenthesis structure

Directed acyclic graphs can be expressed in a linear parenthesis notation which is convenient for textual processing, computation and analysis.

Example

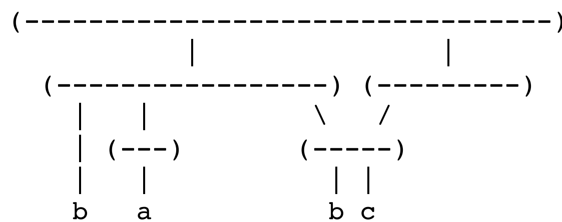
```
V = {1,2,3,4,5,a,b,c}        E = {12,13,24,25,2b,35,4a,5b,5c}

      G = ( ( b ( a ) ( b c ) ) ( ( b c ) ) )
          1 2   4     5         3 5
```





Structure sharing in graphs is represented by labeling:

```
      G = ( ( b ( a )    5   ) (      5    ) )
      5 = (b c)
```



Since parenthesis structure represents both lists and trees naturally, the parenthesis structure of graphs provides a uniform representation for most common CS data structures (except arrays).

Primary graph algorithms

searching

The search algorithm traverses a graph, usually identifying the shortest (or longest) path between two vertices. When the graph has a source (a root, or beginning, vertex) and a sink (a set of terminal vertices), beginning and ending points are defined by those vertices. A variant of the path problem when there is no preferred start or finish is to find the shortest (longest) path between any pair of vertices.

Spanning trees list paths that connect all vertices. The spanning tree problem is to identify the smallest (largest) number of edges which span the graph. The travelling salesman problem (i.e., what is the shortest route available to visit a collection of sites) is an example of computing a minimal spanning tree.

A ***Hamiltonian path*** is a path which visits each vertex while not traversing the same edge more than once. This is an example of a graph theory problem little application to CS.

covering

A ***vertex cover*** is a subset of vertices of an undirected graph such that every edge has at least one end in the subset. The covering algorithm visits every vertex, identifying the minimal number of vertices in the covering subset. Spanning trees necessary cover a graph, but not minimally. Every alternate vertex in a spanning tree also forms a cover.

Minimization of Boolean expressions (by Karnaugh maps or by algebraic techniques) is an example of graph covering. Each product term in a sum-of-products (SOP) representation is a subset of vertices of a Boolean hypercube. The covering problem is to include each vertex in the SOP using a minimum of sub-cubes of the hypercube.

coloring

A ***vertex coloring*** is a labeling of the vertices of an undirected graph such that no edge has two endpoints with the same label. The coloring algorithm visits every vertex, identifying the minimal number of colors necessary to label every vertex.

An example of graph coloring is scheduling and resource allocation using a ***resource conflict graph***. Vertices in a resource conflict graph represent operations to perform; edges represent pairs of operations which are in conflict because they cannot use the same resource. (A ***resource compatibility graph*** is the complement of the conflict graph.)

Graph searching and shortest paths algorithms

Breadth first search

```
BFS(G) =def=
    for each vertex v excluding the starting-vertex do
        color[v] := white;
        distance[v] := infinite                  /* initialization
    color[starting-vertex] := gray
    distance[starting-vertex] := 0
    queue := starting-vertex
    while not[null queue] do
        v := head[queue]
        for each u in adjacent[v] do
            if color[u] = white
                then color[u] := gray
            distance[u] := distance[v] + 1
            push[queue,u]                  /* always gray in queue
            pop[queue]
            color[v] := black
```

Depth first search

```
DFS(G) =def=
    for each vertex v do
       color[v] := white
    time := 0
    for each vertex v do
       if color[v] = white
        then dfs-visit[v]

DFS-VISIT(v) =def=
    color[v] := gray
    time := time +1
    gray-time[u] := time
        for each u in adjacent[v] do
           if color[u] = white
               then dfs-visit[u]
    color[v] := black
    time := time +1
    done[time] := time
    store-topological-sort[v]
```

"Colors" are used to identify the process state of a graph. Below, white refers to *not-yet-processed*, gray to *currently-on-the-frontier*, and black to *done-processing*.

This algorithm outline can be generalized by assigning each vertex a *visit-time*, which then identifies a depth ordering for the vertices. Depth-first visit-times thus count the path length from the start node to each other node. A topological sort can be stored by recording this timing information, providing an easy data structure for computing the length of all traversal paths.

Depth-first search is O(V+E) for an adjacency list representation and O(V^2) for an adjacency matrix representation.

Shortest path questions can be converted into longest path questions simply by reversing the sign of weights for each edge.

Interestingly, ***binary search trees*** are a way to organize data so that search is O(lnN), i.e. proportional to the depth of the tree. Graphs are thus most costly to search. However, since we are at worst using linear algorithms, this is rarely a problem.

Dijkstra's minimal weighted path algorithm

```
Path[G(V,E,w)] =def=
    s[0] = 0;                   /* s is the path
    for i=1 to n do
        s[i] = w[0,i]           /* initialize path weights
    repeat
        select an unmarked vertex v[q] so that s[q] has a minimal weight;
        mark v[q];
        foreach unmarked vertex v[i] do
            s[i] = min[s[i], s[q]+w[q,i]];
    until all vertices are marked.
```

Relaxation

```
init-min-path[G,s] =def=        /* s is the starting vertex
    for each vertex do
        d[v] = infinity;        /* the estimated value to be successively reduced
        path[v] = nil
    d[s] = 0

edge-relax[u,v,w] =def=
    if d[v] > d[u] + w[u,v]
        then d[v] = d[u] + w[u,v]
            path[v]=u

dag-shortest-path[G,w,s] =def=
    topological sort G;
    init-min-path;
    for each vertex in sorted order do
        for each vertex in adjacent[v] do
            relax[u,v,w].
```

The general technique of *relaxation* consists of testing whether a result can be improved by a particular choice by trying that choice. The choice is accepted if it is an improvement, or revoked if it is not an improvement. The upper bound on a result is thus continually decreased. Consider the shortest path algorithm as an example:

Minimum Spanning Tree

```
MST[G,w] =def=
    A := 0;
    while A does not form a spanning tree do
        find an edge [u,v] that is part of the spanning tree
        A := A union [u,v];
    return A
```

Vertex cover

Graphs may provide multiple paths between vertices. The minimal vertex cover is the smallest number of vertices which include at least one endpoint of each edge. The algorithm below is heuristic, providing a cover but not necessarily the minimal cover. Finding the minimal cover is intractable, O(2^n).

```
Vertex-cover-Vertex[G(V,E)] =def=
    C = 0;                          /*initialize cover
    while E =/= 0 do
        select a vertex v;
        delete v from V;            /* remove edges incident with v also
        C = C union v.

Vertex-cover-Edge[G(V,E)] =def=
    C = 0;
    while E =/= 0 do
        select an edge e[u,v];
        C = C union {u,v};
        delete u and v from V.      /* remove edges incident with u and v also
```

Vertex coloring

A *vertex coloring* of a graph is a labeling of the vertices such that no edge has two vertex endpoints with the same label. The optimization problem is to identify the minimal number of colors for a graph. The algorithm below is heuristic; find the minimal coloring is intractable.

```
Vertex-coloring[G(V,E)] =def=
    for i=1 to V do
        c = 1;                      /* c is the number of colors
        while there exists a vertex adjacent to v[i] with color c do
            c = c + 1;
        label v[i] with color c.
```