CE204

Data Structures and Algorithms

Part 7

Graphs 1

A ***graph*** is a collection of vertices and edges, where each edge connects two of the vertices. We shall assume that the two vertices must be distinct (i.e. an edge cannot connect a vertex to itself) and that there is at most one edge connecting any given pair of vertices. An edge connecting two vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ may be denoted by $(\boldsymbol{u},\boldsymbol{v})$.

In an ***undirected*** graph the edge $(\boldsymbol{u},\boldsymbol{v})$ is the same as $(\boldsymbol{v},\boldsymbol{u})$, but in a ***directed*** graph each edge has a direction, so $(\boldsymbol{u},\boldsymbol{v})$ and $(\boldsymbol{v},\boldsymbol{u})$ would be distinct.

A ***path*** from vertex $\boldsymbol{v}_1$ to $\boldsymbol{v}_k$ is a sequence of edges of the form $(\boldsymbol{v}_1,\boldsymbol{v}_2)$, $(\boldsymbol{v}_2,\boldsymbol{v}_3)$ …$(\boldsymbol{v}_{k-1},\boldsymbol{v}_k)$.

Graphs 2

A graph is ***connected*** if for every pair of vertices ***u*** and ***v*** there is a path from ***u*** to ***v***.

A ***cycle*** is a non-empty path from a vertex to itself comprising distinct edges.

A graph that contains no cycles is said to be ***acyclic***.

In a ***weighted*** graph a numeric value (the weight), usually an integer, is associated with each edge. In most applications this will represent some measure of distance, cost or capacity. An edge from ***u*** to ***v*** with weight 5 be may be denoted by (***u***,***v***,5).

Graphs 3

It is common to use n and e to denote the number of vertices and edges respectively of a graph.

It is easy to see that for any graph $e<n^2$, and for any undirected graph $e<n^2/2$.

For a connected graph it is not difficult to see that $e \geq n-1$.

For an acyclic graph it can be shown that $e<n$.

Combining the last two results we observe that if a graph is both connected and acyclic e must be equal to $n-1$.

Implementing Graphs 1

There are many different data structures that can be used to represent graphs. The simplest approach is to store all of the edges in a list, but this is not particularly suitable for most applications since in order to answer questions such as "is there an edge (***u***,***v***)?" and "find all vertices ***x*** such that there is an edge (***u***,***x***)" it would be necessary to traverse the whole list.

The two most common methods for representing graphs are adjacency matrices and adjacency lists – the former optimises the time taken to answer the first question above whereas the latter optimises the time taken to answer the second question.

Implementing Graphs 2

The simplest form of ***adjacency matrix*** is a two-dimensional array of boolean values in which `a[u][v]` will have the value `true` if and only if an edge (***u***,***v***) exists.

If the vertices have non-numeric labels it is necessary to provide a lookup table to match the vertex names to the array indices.

For a weighted graph `a[u][v]` would hold the weight of the edge (***u***,***v***) if the edge exists and some value which is not a valid weight (e.g. 0 or –1) if the edge does not exist.

Implementing Graphs 3

In Java a two-dimensional array is implemented as an array of arrays. Since all of the arrays in the two-dimensional array are of the same size we can create such an array with a single statement:

```
boolean[][] a = new boolean[n][n];
```

Implementing Graphs 4

The ***adjacency list*** method for representing graphs uses for each vertex a list of its neighbours, together with the weights of the edges if appropriate. References to these lists may then be placed in an array or master list.

An adjacency matrix contains $\boldsymbol{n}^2$ entries whereas an adjacency list uses about $\boldsymbol{n}+\boldsymbol{e}$ objects (or $\boldsymbol{n}+2\boldsymbol{e}$ for an undirected graph since information about each edge appears twice). For sparse graphs, where $\boldsymbol{n}$ is large and $\boldsymbol{e}$ is much smaller than $\boldsymbol{n}^2$, the list approach will make the most efficient use of memory, whereas for dense graphs the matrix approach may be better, since the array entries use less space than the objects of the list approach.

Spanning Trees

A ***spanning tree*** for a connected undirected graph is an acyclic connected graph whose vertices are those of the original graph and whose edges form a subset of those of the original graph.

Since it is both connected and acyclic, a spanning tree for a graph with ***n*** vertices must contain exactly ***n***-1 edges (see slide 13).

The ***cost*** of a spanning tree for a weighted connected graph is the sum of the weights of all of the edges in the spanning tree.

Minimum Cost Spanning Trees

Most graphs will have many spanning trees, each with a different cost; a ***minimum cost spanning tree*** (MCST) is one whose weight is less than or equal to the weight of all other possible spanning trees.

The desire to find minimum cost spanning trees arises in a number of applications where the cost reflects some real concept of expense. There are two commonly used algorithms for finding MCSTs; these were developed by Prim and Kruskal. For some graphs Prim's algorithm is more efficient whereas for others Kruskal's algorithm performs better, so the choice of which to use should be made with the aid of some knowledge about the graph.

MCSTs: Prim's Algorithm 1

The basic idea behind ***Prim's algorithm*** is to choose any vertex as a starting point and then gradually build a spanning tree by adding edges with minimal weights. At all intermediate stages the spanning tree being built will be connected.

We can describe the algorithm abstractly using pseudo-code as shown on the next slide; the variable `n` refers to the number of vertices. Since the number of edges in a spanning tree for a graph with ***n*** vertices is ***n***-1 and one edge is added each time the loop body is executed, this body must be executed precisely ***n***-1 times.

MCSTs: Prim's Algorithm 2

```
initialise the MCST to have no edges;
create an empty set of vertices V;
insert any vertex from the graph into V;
for (int i = 0; i<n-1; i++)
{ let e be the lowest cost edge connecting a
    vertex in V to a vertex w that is not in V;
  add e to the MCST;
  insert w into V;
}
```

MCSTs: Prim's Algorithm 3

In order to implement Prim's algorithm efficiently we need to be able to find the lowest cost edge connecting a vertex in `v` to a vertex not in `v` without having to search all of the edges. Hence it is not appropriate to actually represent `v` as simply a set of vertices.

Instead we maintain a table containing for each vertex its nearest neighbour (i.e. the one with the lowest-cost edge) in `v` (if it has one) and the weight of the edge connecting the vertex to that neighbour. A weight of 0 could be used to indicate that the vertex is itself in `v` and a weight of -1 to indicate that there are no neighbours.

MCSTs: Prim's Algorithm 4

Initialisation of the table involves setting the entry for the chosen starting vertex to be 0 and setting all of the other entries to -1, then updating the entries for the neighbours of the starting vertex.

Inside the loop body after adding an edge to the MCST we need to check and, if necessary, update the entries for all of `w`'s neighbours. This can be done efficiently if the original graph is represented using the adjacency list method.

MCSTs: Prim's Algorithm 5

Finding the lowest-cost edge connecting a vertex in **v** to a vertex not in **v** simply involves finding the smallest positive entry in the table and can be done in O(***n***) time. Updating involves traversing a list containing less than ***n*** entries and can be also done in O(***n***) time. Hence the time taken for the loop body is O(***n***) and the time taken for the whole loop is O($\boldsymbol{n}^2$) . Initialisation can be performed in O(***n***) time so the time complexity for Prim's algorithm is O($\boldsymbol{n}^2$).

MCSTs: Kruskal's Algorithm 1

Kruskal's algorithm builds a minimum cost spanning tree by selecting lowest-cost edges from anywhere in the graph instead of building outwards from a single vertex. Hence the MCST is not necessarily connected at intermediate stages. We can express the algorithm in pseudo-code as seen on the next slide.

We cannot tell how many times the loop body will be executed since on some occasions no edge is added to the MCST. Hence we need to maintain a count of the number of edges that have been added and continue until this reaches ***n***-1.

MCSTs: Kruskal's Algorithm 2

```
sort the edges of the graph by weight
   (smallest first) into a list L;
initialise the MCST to have no edges;
int i = 0;
while (i<n-1)
{ Edge e = L.head();
   if (the MCST with e added would be acyclic)
   { add e to the MCST;
      i++;
   }
   L = L.tail();
}
```

MCSTs: Kruskal's Algorithm 3

The key to the efficient implementation of Kruskal's algorithm is to be able to easily determine whether adding an edge to the MCST will introduce a cycle. To facilitate this we maintain a collection of ***sets of connected components*** (i.e. sets of vertices that form connected sub-graphs of the incomplete spanning tree). If the two vertices of a new edge are in the same set there is already a path between them so adding this edge would complete a cycle.

MCSTs: Kruskal's Algorithm 4

Initially there are no edges in the spanning tree so each vertex should be placed in a separate connected-component set. When a new edge is added to the spanning tree the two connection sets containing its vertices must be merged into a single set.

We can store the sets as lists and use an array containing, for each vertex, a reference to the set containing that vertex. Merging of two sets involves moving the elements of one list into another and also updating the array entries for these elements.

MCSTs: Kruskal's Algorithm 5

The main time-consuming activities in Kruskal's algorithm are the initial sorting of the edges and the updating of the connection set information.

Using an algorithm such as mergesort we can perform the sorting in O($\boldsymbol{e}$ log $\boldsymbol{e}$) time.

If we ignore the time for merging of connection sets the time for the loop body is O(1) and since the maximum number of times the loop body is executed is $\boldsymbol{e}$, the time for the loop is O($\boldsymbol{e}$) so the total time for all activities apart from merging of connection sets is O($\boldsymbol{e}$ log $\boldsymbol{e}$).

MCSTs: Kruskal's Algorithm 6

We need to add to the O(***e*** log ***e***) time obtained on the previous slide the total time for the merging of connection sets. The merging of two connection sets could potentially take time proportional to ***n***, and there are ***n***-1 merges, so the worst-case total time for all of the merges is O($\boldsymbol{n}^2$). However, if we ensure that when merging we always move the elements of the smaller set into the larger set, then it can be shown that no vertex will be moved more than $\log_2 \boldsymbol{n}$ times, so (since there are ***n*** vertices) the total time for merging will be reduced to O(***n*** log ***n***).

Since for a connected graph with cycles $\boldsymbol{e} \geq \boldsymbol{n}$, we can conclude that if implemented efficiently the time complexity for Kruskal's algorithm is O(***e*** log ***e***).

Decision-Making: Which Algorithm to Use

We have seen that the time complexity for for Prim's algorithm is $O(n^2)$, whereas Kruskal's algorithm is $O(e \log e)$. We observe that one of the complexities depends on the number of vertices and the other depends on the number of edges.

For very dense graphs where e is much larger than n, $e \log e$ will be significantly larger than n^2 so Prim's algorithm will give the best performance; for large sparse graphs, where e is relatively close to n, $e \log e$ will be much smaller than n^2 so Kruskal's algorithm will perform better.

Shortest Paths: Dijsktra's Algorithm 1

In applications using weighted directed or undirected graphs it is often necessary to find the ***shortest path*** from one vertex to another.

Dijkstra's algorithm determines the shortest paths from a given vertex to all other vertices for which paths exist; if we require the shortest path to a given vertex we can stop as soon as that has been found. If the algorithm is applied to a connected undirected graph the edges in the paths found will form a spanning tree, but this will not necessarily be a minimum cost spanning tree.

Shortest Paths: Dijsktra's Algorithm 2

The algorithm discovers shortest paths in ascending order of distance. As it proceeds we maintain two sets of vertices: the ***known set*** contains all vertices to which the shortest path has been found and the ***frontier set*** contains all other vertices to which there is an edge from one of the elements of the known set.

When a vertex is in the frontier set a path to it has been found, but it is not yet known if this is the shortest path.

Shortest Paths: Dijsktra's Algorithm 3

At each step of the algorithm the element of the frontier set with the shortest distance from the starting point is moved to the known set, and the frontier set is updated.

To be able to find this element we need to maintain for each vertex the length of the shortest path (if any) yet found. To keep track of the paths that have been found we must also store details of the penultimate vertex in that path.

In the pseudo-code presentation of the algorithm on the following slides this information is assumed to be stored in two arrays called `dist` and `prev`.

The algorithm terminates when the frontier set is empty.

Shortest Paths: Dijsktra's Algorithm 4

```
Set known = {s} frontier = {};
   // s is start vertex
dist[s] = 0;
for all edges (s,v,w)   // w is the weight
{ insert v into frontier
  dist[v] = w;
  prev[v] = s;
}
// continued on next slide
```

Shortest Paths: Dijsktra's Algorithm 5

```
while (frontier is not empty)
{ let x be the element of frontier whose
        dist value is the smallest
  move x from frontier to known
  for all edges (x,v,w)
  { if (v is in frontier)
    { int vdist = dist[x]+w;

      if (vdist<dist[v])
       { dist[v] = vdist; prev[v] = x;
       }
    }
  // for loop body continued on next slide
```

Shortest Paths: Dijsktra's Algorithm 6

```
  // for loop body continued
    else if (v is not in known)
    { insert v into frontier
      dist[v] = dist[x]+w;
      prev[v] = x;
    }
  }
}
```

Shortest Paths: Dijsktra's Algorithm 7

When the algorithm terminates the distances in the table will be the distances of shortest paths to each vertex in the known set, and we can find the path to a vertex by working backwards to previous vertices.

If the algorithm has terminated and any vertex is not in the known set there is no path from the starting point to that vertex.

To be able to efficiently determine whether `v` is in the known or frontier sets we can use an additional array indicating to which set each vertex belongs. There is then no need to store the known set explicitly.

Shortest Paths: Dijsktra's Algorithm 8

Using the approach described on the previous slide to check whether `v` is in the known or frontier sets the time for the for loop body is O(1). If an adjacency list representation is used for the graph we can easily find all edges `(x,v,w)` and since there are at most ***n***-1 such edges the total time for the for loop will be O(***n***).

Finding and moving `x` at the start of the while loop also takes O(***n***) time so the while loop body is O(***n***). Since each iteration adds a vertex to the known set the body is executed at most ***n***-1 times, so the time for the while loop is $O(n^2)$.

Initialisation takes O(***n***) time so the time complexity of the algorithm is $O(n^2)$.