Information Technology Mathematics

Trees

Lecturer: Markus Hofmann

Lecture Outline

Continuing with graphs & trees, basic concepts and how they can model structures in computing

- Trees
- Applications of trees
- Spanning trees
  - Minimal spanning trees
- Minimum distance paths
- Rooted trees
- Binary trees
- Tree traversal: in-order, pre-order, post-order

Trees

- Definition
  - A tree is a connected graph with no cycle

  Remember a *cycle* is path is a graph that:
    - includes at least one edge,
    - there are no repeated edges,
    - the first and last vertex coincide but there are no other repeated vertices

    i.e. A circuit with some additional properties

  - Any tree with n vertices has n – 1 edges.

  Examples of trees – (on board)

Spanning trees

- *Weighted graph* – a weighted graph is a simple graph in which each edge has a positive number attached to it. This number attached to the edge in a weighted graph is called the *weight* of the graph
- e.g. using graphs in the design of a LAN
  - Suppose have a site (college campus) which has several buildings with computer labs
  - Want to build a communication network – a LAN, between buildings
  - Can use a graph to represent this where the vertices are the buildings and the edges the link between the buildings
  - The cost of linking each pair of buildings or say the capacity of the link etc. can be represented by a weight (number) on the edge
  - Example weighted graph (on board)

Spanning trees

- In building the LAN – we want to create links in such a way that it is possible for a computer in any lab to communicate with a computer in any other lab (doesn’t mean that every pair of buildings must be directly linked – it is acceptable for two buildings to be linked indirectly via a third for instance)
- Example matrix representing cost of linking each pair of buildings (on board)

- Installing a link between two buildings costly so would like to build the network as cheaply as possible
- We do not need to install all the links just the three cheapest

- So could install CD, BC and AC
  - We these links in place any two labs can communicate with one another.
- Small problem – so could solve by inspection
- What if we wanted to solve this for a large number of sites? (not practical to try solve it in a trial and error way
  - Need an algorithm to find the cheapest network linking the sites

- A first step towards finding a solution – look for some properties that the solution must have
  - The solution itself must be a graph
  - Consist of all the vertices of the original graph but only some of the edges of the original graph
  - Must be a connected graph for any two sites to communicate
  - Cannot contain any cycles
- Solution is a *Tree* – **Spanning Tree**
  - where the vertices of the tree are the vertices of the original graph and
  - the edges of the tree must be selected from the set of edges in the original graph

Minimal Spanning Tree

- A connected graph can have many spanning trees
  - For previous example could also have AB, AC and AD or the edges AB, BC and CD
- Remember we want to install just the three cheapest links i.e. *the smallest possible total weight*
- A spanning tree with this property is called a **Minimal Spanning Tree (MST)**

- Can state this problem more formally: as design an algorithm that inputs a connected <u>weighted</u> graph and outputs a <u>minimal spanning tree</u>

Minimal Spanning Trees

- How to find the MST
  - Start at one of the vertices
  - Pick an edge with the least weight of all the edges incident to that vertex
    - This edge forms the start of our MST
  - Next look for an edge with least weight of all the edges (that does not form a cycle) that join up with the edge we already have
  - Continue this way adding on edge at a time to obtain a MST
  - Note when working through this process, if the addition of an edge would make a cycle then we don't want that edge (tree structure should not have a cycle)

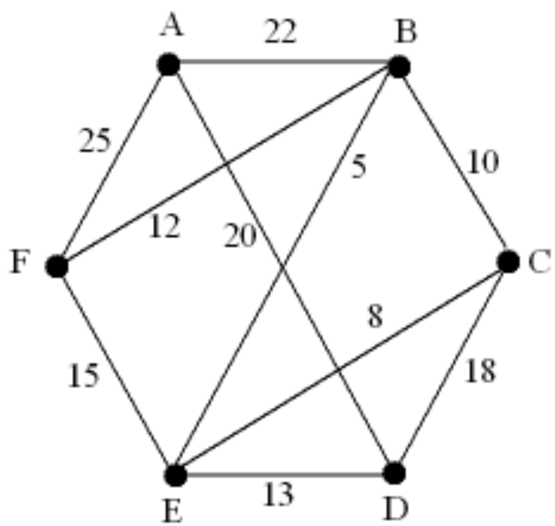
Minimal Spanning Trees

- More formally can write this as <u>*Prim's algorithm*</u>

1. Input a connected weighted graph *G*, with vertex set $V(G) = \{v_1, \ldots, v_n\}$ and edge set $E(G) = \{e_1, \ldots, e_m\}$
2. *T* = $\{v_1\}$; *unused_edges* = *E(G)*
3. **For** *i* = 1 to *n-1* **do**
   - 3.1 *e* = the first edge with minimal weight in *unused_edges* that is incident to exactly one vertex in *T*
   - 3.2 *v* = the vertex not in *T* to which *e* is incident
   - 3.3 *T* = *T* U {*e, v*}
   - 3.4 *unused_edges* = *unused_edges* – {*e*}
4. Output *T*

Note: For-Do loop executed n-1 times (where n is the number of vertices in the weighted graph) as any tree with n vertices has n-1 edges and a spanning tree for a graph has the same number of vertices as the original graph

- Example – use *Prim's algorithm* to find a minimal spanning tree for the weighted graph shown below



<u>Solution:</u>

- Starting at vertex *A*, find that the edge with least weight incident to *A* is the edge *AD* with weight 20, so this is the first edge in the minimal spanning tree *T*

- Look next at the edges incident to both *A* and *D* and not yet in *T*.

- Of these *DE* has the least weight so we add it to *T*

- Continue in a similar fashion, we add *EB* and *EC* to *T*.

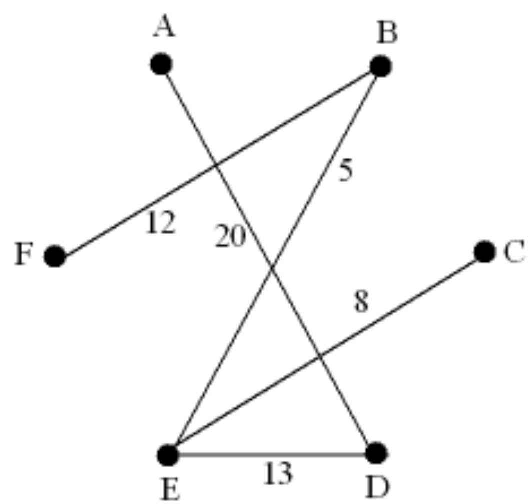
- At this stage the edge with the least weight incident to vertex *T* is *BC* with the weight 10 but this edge is to two vertices of *T* and therefore we cannot not add it to *T*

- The edge with the least weight that is incident to exactly one vertex of *T* is *BF*, so this is the edge to be added.

MST

- The resulting minimal spanning tree is shown below which has a total weight of 58

The tree *T* has a total of 5 edges
Since the original graph has 6 vertices and 5 = 6-1 there are no more edges to be added



MST – Trace table for the previous algorithm

Step	***e***	***v***	**T**	***unused_edges***	**Output**
2	**-**	**-**	**{A}**	**{AB, AD, AF, BC, BE, BF, CD, CE, DE, EF}**	**-**
3.1 - 3.4	**A D**	**D**	**{A, AD, D}**	**{AB, AF, BC, BE, BF, CD, CE, DE, EF}**	**-**
3.1 - 3.4	**D E**	**E**	**{A, AD, D, DE, E}**	**{AB, AF, BC, BE, BF, CD, CE, EF}**	**-**
3.1 - 3.4	**B E**	**B**	**{A, AD, D, DE, E, BE, B}**	**{AB, AF, BC, BF, CD, CE, EF}**	**-**
3.1 - 3.4	**C E**	**C**	**{A, AD, D, DE, E, BE, B, CE, C}**	**{AB, AF, BC, BF, CD, EF}**	**-**
3.1 - 3.4	**B F**	**F**	**{A, AD, D, DE, E, BE, B, CE, C, BF, F}**	**{AB, AF, BC, CD, EF}**	**-**
4	**B F**	**F**	**{A, AD, D, DE, E, BE, B, CE, C, BF, F}**	**{AB, AF, BC, CD, EF}**	**{A, AD, D, DE, E, BE, B, CE, C, BF, F}**

Weighted graph representation

- How can a weighted graph be represented in a form suitable for machine computation?
  - Use a matrix i.e. a **weight matrix**, similar to the adjacency matrix
  - Where put the weight of the edge from vertex v_i to v_j rather than the number of edges in row *i* and col *j*

  - What goes in if there is no edge form vertex v_i to v_j ?
  - For the type of weighted graph we have just seen the weights represent a penalty of some kind: i.e. cost or distance which we want to minimise in a typical application
    - e.g. travelling salesman problem – a sales representative wants to visit a number of towns and return home, travelling the shortest possible total distance in the process
  - Alternatively the weights could represent we want to maximise – e.g. capacity of a communication channel

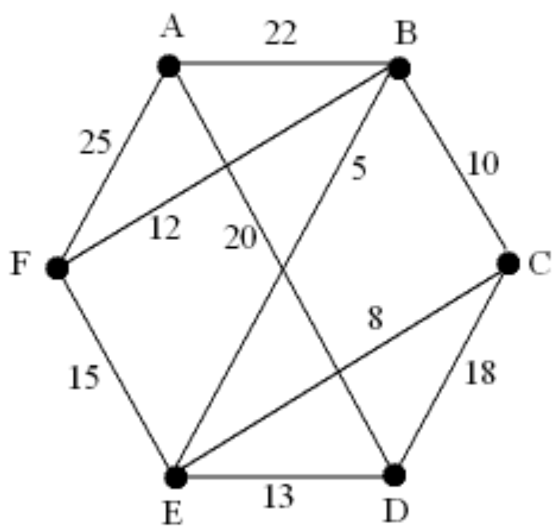
Weighted graph representation

- Let G be a weighted graph with vertices $v_1, v_2, \ldots, v_n$, in which the weights represent a penalty of some kind. The weight matrix of G is the n x n matrix for which the entry w_{ij} in the ith row and jth col is given by the rule:

$$w_{ij} = \begin{cases} 0 \textbf{ if } i = j \\ \textbf{the weight of the edge from } i \textbf{ to } j \textbf{ if } v_i \textbf{ and } v_j \textbf{ are adjacent} \\ \infty \textbf{ if } i \neq j \textbf{ and } v_i \textbf{ and } v_i \textbf{ are not adjacent} \end{cases}$$

Weighted graph representation

- Write down the weight matrix for this graph:



$$
\begin{array}{c c}
\begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} &
\begin{bmatrix}
0 & 22 & \infty & 20 & \infty & 25 \\
22 & 0 & 10 & \infty & 5 & 12 \\
\infty & 10 & 0 & 18 & 8 & \infty \\
20 & \infty & 18 & 0 & 13 & \infty \\
\infty & 5 & 8 & 13 & 0 & 15 \\
25 & 12 & \infty & \infty & 15 & 0
\end{bmatrix} \\
 & \begin{array}{cccccc} A & B & C & D & E & F \end{array}
\end{array}
$$

Minimal distance paths

- Weighted graph could represent the communications network in looked at earlier but this time let the weight of each edge represent the time taken by a signal to travel along the link in the network

- Finding the quickest (or shortest) path through a graph (representing some situation) of vertices is a considerable problem in computing
  - In this example we want to find the quickest path for a signal to take through the network from one vertex to the other.

Minimal distance paths

- Let G be a connected graph, and let u and v be vertices in G. If P is a path from u to v, then the length of P is the sum of the weights of the edges in P. The distance from u to v is the smallest of the lengths of all the paths from u to v.
- Can use $d(u, v)$ to denote the distance from u to v
- And $weight(e)$ to denote the weight of an edge e

- Dijkstra's algorithm provides an efficient solution to the minimum distance problem

Dijkstra's algorithm

1. Input a connected weighted graph G with vertex set $V(G) = \{v_1, \ldots, v_n\}$ and edge set $E(G) = \{e_1, \ldots, e_m\}$
2. *$T = \{v_1\}$; $d(u, v) = 0$*
3. **For** $j = 1$ to $n-1$ **do**

   3.1 w = the first vertex for which $d(v_1, w) + weight(e)$ is minimised. Where w ranges over all the vertices in T, and e ranges over all the edges in $G - T$ that are incident to w

   3.2 e = the first edge incident to w for which $d(v_1, w) + weight(e)$ is minimised

   3.3 v = the vertex in $G\text{-}T$ to which e is incident

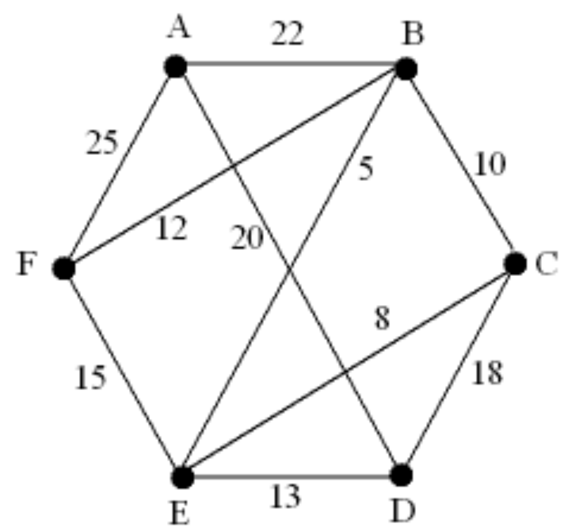
   3.4 $T = T \cup \{e, v\}$

   3.5 $d(v_1, v) = d(v_1, w) + weight(e)$

   4. Output T

Dijkstra's algorithm

- Example use Dijkstra's algorithm to find the minimum distance from A to C in the weighted graph below and the path that achieves this distance.



Dijkstra's algorithm

Solution:

Step	*w*	*e*	*v*	T	$d(v_1, v)$	Output
2	**-**	**-**	**-**	**{A}**	**-**	**-**
3.1 - 3.5	**A**	**AD**	**D**	**{A, AD, D}**	**20**	**-**
3.1 - 3.5	**A**	**AB**	**B**	**{A, AD, D, AB, B}**	**22**	**-**
3.1 – 3.5	**A**	**AF**	**F**	**{A, AD, D, AB, B, AF, F}**	**25**	**-**
3.1 - 3.5	**B**	**BE**	**E**	**{A, AD, D, AB, B, AF, F, BE, E}**	**27**	**-**
3.1 - 3.5	**B**	**BC**	**C**	**{A, AD, D, AB, B, AF, F, BE, E, BC, C}**	**32**	**-**
4	**B**	**BC**	**C**	**{A, AD, D, AB, B, AF, F, BE, E, BC, C}**	**32**	**A, AD, D, AB, B, AF, F, BE, E, BC, C}**

The distance from *A* to *C* is 32 and the path is *ABC*

Rooted Trees

- Rooted tree – one of the vertices is specified as the *root*
  - Tree is drawn with the root at the top
- Vertices adjacent to the root, ‘first generation’, are shown below the root
  - i.e. in a horizontal line below the root
- Vertices below these (i.e. can be reached from the root by a path of length 2), the ‘second generation’, are shown below the first generation and so on…
- Example (on board)

Rooted Trees

- Family tree is an example of a rooted tree (similar terminology associated with family tree is used with rooted trees)

Terminology:

- *Parent* vertex – vertex immediately above a given vertex
- *Child* vertex – vertex immediately below a given vertex
- *Leaf* – a vertex with no children
- Every vertex in a rooted tree has exactly one parent
- A vertex in a rooted tree is an *ancestor* or a *descendant* of another

Rooted Trees

- Rooted trees are widely used in computing to represent a decision process i.e. <u>decision trees</u>
- Example of a decision tree: suppose want to sort three distinct numbers denoted by a, b and c into increasing order.
  - Take two numbers at a time and compare them
  - More comparisons may be needed…

- Can depict this procedure using a rooted tree (on board)
- The decision process begins at the root and moves down the tree until a leaf is reached
  - where the leaves represent the final arrangements of the three numbers into increasing order and the other vertices represent points at which a decision is made

Binary tree

- *Binary* rooted tree – every vertex that is not a leaf has exactly two children
- The two children of each parent are the *left child* and the *right child*
- Left child – the root of the *left subtree*
- Right child – the root of the *right subtree*

- Binary trees can be use to represent algebraic expressions

Binary tree

- Given the expression (a – (b/c)) x (d + e)
  - To evaluate using a computer with values substituted in for the variables (on board)
- The *principle expression* in this example, is the multiplication (it is executed last after (a – (b/c)) and (d + e) have been evaluated , remember precedence from last year!)
- The principle operations of these sub-expressions are subtraction and multiplication
- (a – (b/c)) also contains the sub-expression (b/c) with division as its principle operation

- This can all be depicted using binary rooted tree (called an expression tree)
  - Where the root is labelled with the principle expression
  - Left sub-expression is represented by the left subtree (which has a right subtree of its own)
  - Right sub-expression is represented by the right subtree

In-order traversal

- A computer can process this expression by visiting the vertices of the expression tree in a particular order - *traversal* of the tree
- In-order traversal (infix)
  - Visit all of the vertices of the left sub-expression
  - Then visit the root
  - And lastly visit the vertices of the right sub-expression

- Algorithm *in-order_traverse(T)*

  1. If *T* is not a leaf then

     1.1 *in_order_traverse* (left subtree of *T*)

  2. Output the root of *T*
  3. If *T* is not a leaf then

     3.1 *in-order_tranerse* (right subtree of *T)*

In-order traversal

- Example – apply in-order traversal to the expression tree for (a – (b/c)) x (d + e)
- (on board)

Pre-order Traversal

- Pre-order traversal – visit the root before visiting each subtree
- Algorithm *pre-order_traverse(T)*

1. Output the root of *T*
2. If *T* is not leaf then

   2.1 *pre-order_traverse* (left subtree of *T*)

   2.2 *pre-order_traverse* (right subtree of *T*)

Example – carry out the pre-order traversal of the expression tree for (a – (b/c)) x (d + e) (on board)

Post-order Traversal

- Post-order traversal – visit the root after visiting each subtree
- Algorithm *post-order_traverse(T)*

1. If *T* is not a lea then
   1.1 *post-order_traverse* (left subtree of *T*)
   1.2 *post-order_traverse* (right subtree of *T*)
2. Output the root of *T*

Example – carry out the post-order traversal of the expression tree for (a – (b/c)) x (d + e) (on board)

Summary

- Trees are particularly useful in commuting
  - Used to construct networks with the least expensive set of lines linking distributed computers
  - Employed to construct efficient algorithms for locating items in a list
  - Used to construct efficient codes for storing and transmitting data
  - Used to model procedures that are carried out using a sequence of decision
- Tree – graph with no cycle
- Covered some basic concepts if trees – minimal spanning trees, finding the shortest path, rooted trees and applications of trees such as binary search trees, traversal of trees for computation …