MATH1510 - Discrete Mathematics
Trees

University of Newcastle

UoN

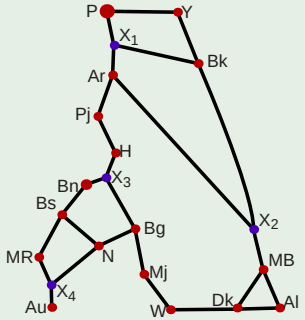
Outline

Spanning subgraphs and spanning trees

For any graph G, a Spanning Subgraph of G is a subgraph that contains all the vertices of G. A Spanning Tree of G is a tree T that is a spanning subgraph of G.

Example

Find some spanning subgraphs and a spanning tree of this graph



Existence of spanning trees

Theorem

Every connected graph has a spanning tree.

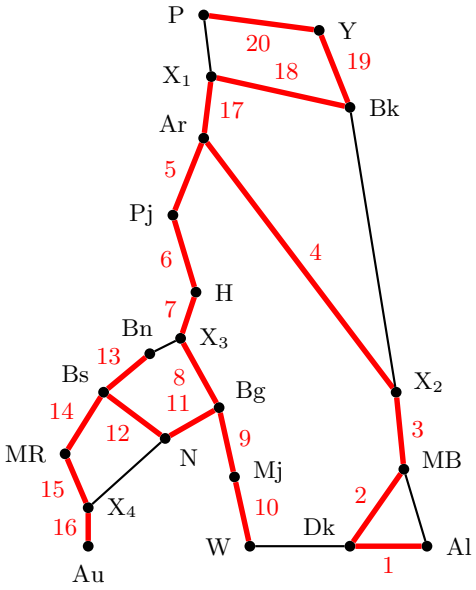
Proof.

Let G be a connected graph that is not already a tree. Then G must contain a cycle. Choose any edge belonging to this cycle, and remove it. If the graph is still not a tree, it must still contain a cycle. Repeat the process until there are no cycles left. □

Depth-first search

- Label the vertices.
- List the vertices in some order.
- Take the first vertex as the root.
- Taking the vertices in the order they are given, adjoin the vertices (and their edges) one-at-a-time, each vertex added is adjoined to the most recently used vertex, if possible.
- If not possible to adjoin a vertex to the last-used vertex, backtrack only as far as necessary to reach a vertex where you are able to add another edge.
- Repeat until all vertices are in the tree.

An example

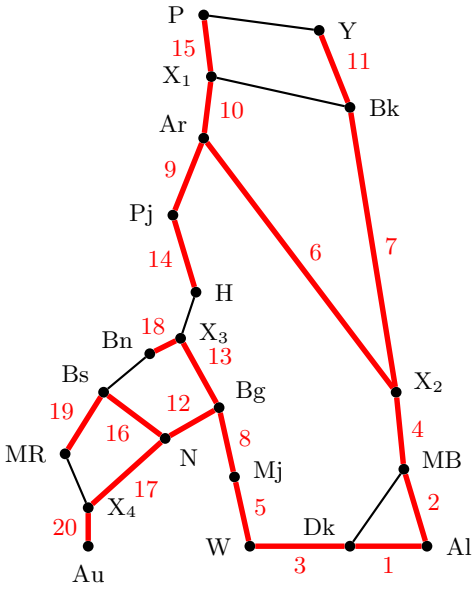


- This is the DFS tree for the alphabetic ordering of the vertices: Al, Ar, Au, Bg, Bk, Bn, Bs, Dk, H, MB, Mj, MR, N, P, Pj, W, X1, X2, X3, X4, Y
- Edge labels indicate the order in which edges are added.

Breadth-first search

- Label the vertices.
- List the vertices in some order.
- Take the first vertex as the root.
- Taking the vertices in the order they are given, adjoin as many as possible to the root.
- Proceed to the first vertex on Level 1 and adjoin as many of the remaining vertices of G as possible to it (being careful not to form a cycle) in the order they are listed.
- Repeat for each of the other Level 1 vertices in order.
- Continue the process for Level 2, and then Level 3, etc. until all vertices are in the tree.

Example

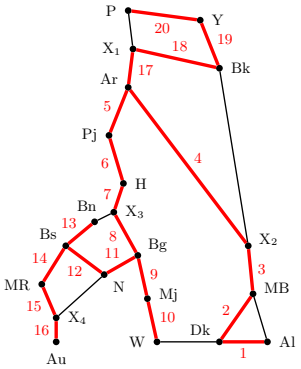


- BFS for the alphabetic ordering of the vertices: Al, Ar, Au, Bg, Bk, Bn, Bs, Dk, H, MB, Mj, MR, N, P, Pj, W, X1, X2, X3, X4, Y
- Edge labels indicate the order in which edges are added.
- Level 1: Dk, MB
- Level 2: W, X_2
- Level 3: Mj, Ar, Bk
- ...

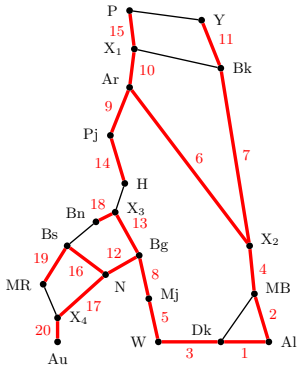
Comparison

Depth-first backtrack as late as possible

Breadth-first backtrack as early as possible



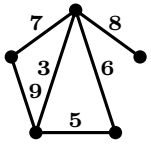
DFS

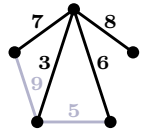


BFS

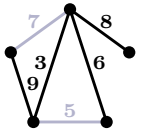
Minimum spanning trees

In a weighted graph, we may need to find a spanning tree in which the total weight is as small as possible. This is a Minimum Spanning Tree (MST).

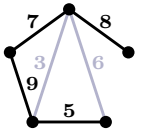




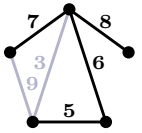
Total wt = 24



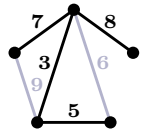
Total wt = 26



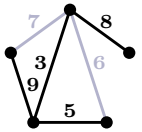
Total wt = 29



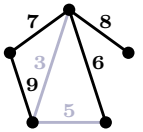
Total wt = 26



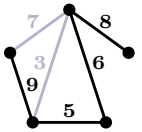
Total wt = 23



Total wt = 25

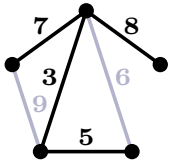


Total wt = 30



Total wt = 28

MST:



Total wt = 23

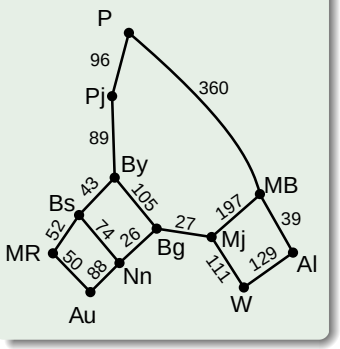
Prim's Algorithm

Prim's algorithm builds a spanning tree T one edge at a time.

- Start with T being an empty edge set and pick any vertex of G (call it s)
- Of all the edges incident with s, choose the one with least weight and adjoin that edge (and its vertex t) to T.
- Of all the edges incident with either s or t, choose the one with least weight and adjoin that edge (and its vertex) to T.
- At each step find the edge of least weight joining a vertex that is already in T with a vertex not in T, and add that edge to T.
- Repeat this process until all vertices of G are in T.

Example

Example



- Step 1: $T = \emptyset$, $s = Au$, shortest edge: $\{Au, MR\}$
- Step 2: $T = \{\{Au, MR\}\}$, shortest edge: $\{Bs, MR\}$
- Step 3: $T = \{\{Au, MR\}, \{Bs, MR\}\}$, shortest edge: $\{Bs, By\}$
- Step 4: ...

Optimality

Theorem

Prim's algorithm finds a minimum spanning tree.

Proof.

(Induction) If we assume that the partial tree constructed after k steps is part of a minimum spanning tree, then we can show that the bigger tree obtained at step $k+1$ by adding the smallest available edge is also part of a minimum spanning tree. □

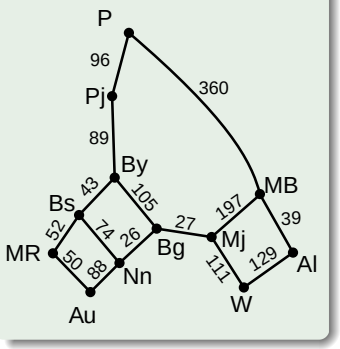
Kruskal's Algorithm

This algorithm also builds the tree one edge at a time.

- List the edges of the graph G in increasing order of weights.
- Start the tree by inserting the first two edges from the list.
- For the next edge in the list: if it forms a cycle with the edges already in the tree, omit it; otherwise insert it into the tree.
- Repeat this for every edge in the list (or until you can see that every vertex is present).

Example

Example



- Step 1: $T = \emptyset$, shortest edge: $\{Bg, Nn\}$
- Step 2: $T = \{\{Bg, Nn\}\}$, shortest edge: $\{Bg, Mj\}$
- Step 3: $T = \{\{Bg, Nn\}, \{Bg, Mj\}\}$, shortest edge: $\{Bs, By\}$
- Step 4: ...

Greedy algorithms

Both Prim's and Kruskal's are examples of greedy algorithms. At each step of a greedy algorithm, you take the best available choice of all that are available.

We have seen that greed is good for minimal spanning trees. However greed can be problematic for other problem types. Greedy algorithms do not always find the optimum overall solution. Generally they suffer from being short-sighted, looking only for "local" optimal features, and so they can easily miss an overall optimal solution.

Example

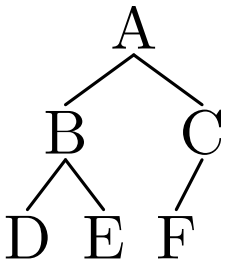
Greedy algorithm for shortest path from Augusta to Albany

Traversals

With data stored in the vertices of a tree, we need ways of accessing them in a well-defined order. This involves finding some systematic "walk-through" or traversal of the tree. For a *binary* tree, there are three natural ways of doing this:

- Preorder: Root, Left, Right
- Inorder: Left, Root, Right
- Postorder: Left, Right, Root

Example



Summary

- Preorder: *ABDECF*
- Inorder: *DBEAFC*
- Postorder: *DEBFCA*

Expression trees

Arithmetic expressions consist of operators and operands. For example,

$$(A + B)/(C * D)$$

- operators: $+$, $-$, $*$, / [These are binary operators]
- operands: A, B, C, D

For any expression, we can store its operands and operators as the vertices of a tree. Then by traversing the tree, we can reconstruct the expression.The *operands* are stored in the *terminal vertices* and the *operators* in the *internal vertices*.
Standard mathematical notation has the operator *between* the operands. This is Infix notation, and corresponds to the inorder traversal of the tree.
Preorder traversal results in Prefix notation.
Postorder traversal results in Postfix notation.

Examples

Construct trees and convert to other notation.

- Infix: $A + B$, $A + B - C$, $A * (B - C)$, $A * B - C$
- Prefix: $-+4a2$, $+-XY/2Z$, $/+AB-XY$
- Postfix: $AB+CDE*/-$, [41] [17] $-$ [2] / [3] [12] [3] / $-$ $-$

Relation to RPN Arithmetic

RPN (postfix) is optimised for stack-based (LIFO) arithmetic. For this reason compilers for languages such as C, C++, Fortran re-encode any arithmetic formulas into stack-based arithmetic. Typically compilers do this by *parsing* a formula into a tree and then producing the postfix version.

This parsing requires a specification of operator precedence and left-associativity vs right-associativity.

BODMAS rules

- Brackets
- Of: includes $2x$ meaning "2 of x", functions like $\sin x$ meaning "sine of x" and also exponents such as 2^3)
- Division and Multiplication: equal priority, in left-associative (i.e. left-to-right) order
- Addition and Subtraction: equal priority, in left-associative order

Examples

Parsing and stack-based arithmetic for

- $10 + 1 - 2 * 3$
- $3 * (24 - 17) - (17 + 13)/(1 + 2 + 3) + 12$

Binary search trees

In computing, it is essential to keep track of the location of data, whether this is

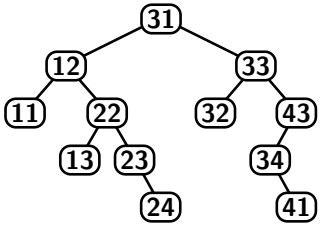
- in the working memory (RAM) of a running program,
- within a file on a disk,
- across several files in a filesystem, or
- in a database.

All of these require a flexible way of accessing and changing the data, and often being able to retrieve it based on some attribute to be looked up. (e.g. a file name, a record number, etc)
Trees provide several ways of organising data in a flexible, efficient ways. We will look at a particular example: Binary Search Trees.

Definition and Properties

If the data to be stored can be arranged in some natural order (such as numerical order or alphabetical order), then it can be stored in the nodes of a Binary Search Tree (BST) for easy access.
To be a binary search tree, a binary tree must satisfy this condition:

For every internal vertex v, all the items in the left subtree of v must be less than the item stored in v, and all the items in the right subtree of v must be greater than the item stored in v.



Some questions

- Where is the smallest item stored in a BST?
- Where is the largest?
- How do you list the items in a BST?
- What does the preorder traversal of a BST look like?
- Inorder & Postorder?

Searching a BST

How do we locate an item in a BST? An algorithm in pseudocode:

```
function FindInTree(tree T, data X) {
  Set here to the root of T
  Repeat {
    If X = here, then stop, data X is found
    If X < here, then
      if leftchild of here exists, then
        set here to leftchild of here
      else stop, data X not found
    If X > here, then
      if rightchild of here exists, then
        set here to rightchild of here
      else stop, data X not found
    }
  }
```

Inserting into a BST

The above algorithm can be modified to give

`function InsertInTree(tree` T`, data` X`)`

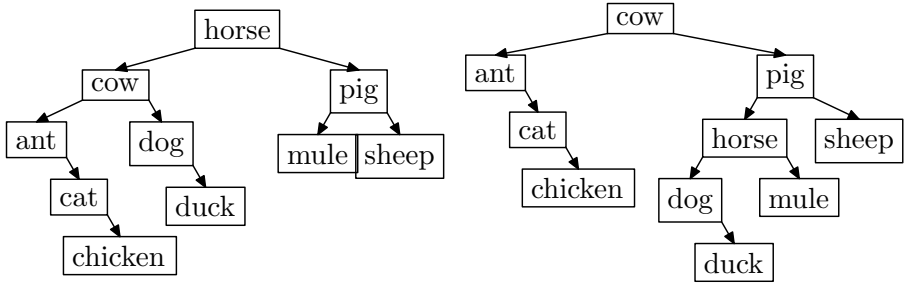
Example

Create a BST holding the following items of data, ordered alphabetically: horse, cow, dog, pig, ant, cat, mule, chicken, duck, sheep

Example

Create a new BST for the same data, inserting them in the following order: cow, pig, horse, ant, dog, cat, mule, chicken, sheep, duck

Examples



Keys

Sometimes the data doesn't have a natural order arrangement, so we need to impose one.
An item of data usually is a record, which is made up of a number of fields, each containing some information. We can simply add another field to the record, called the Key, which contains a number. We then identify the record by its key-value and use the key to order the records. (This is common in databases.)

Efficiency

Question: Is a binary search tree always an efficient way to store data?
Not necessarily - it depends on the shape of the tree.
For a tree containing n vertices (records)

- it may take as many as n steps to locate a record.
- it may take as few as 1
- we may try to minimise the average access time
- we may try to minimise the maximum access time.