# Lecture 8: Trees

## CAB203 Discrete Structures

Matthew McKague

Queensland University of Technology

*matthew.mckague@qut.edu.au*

## Outline

Trees

Algorithms for trees

Case study: Teleportation

Case study: Fifteen

## Readings

This week:

- No readings this week

Next week:

- No readings next week

## Outline

Trees

Algorithms for trees

Case study: Teleportation

Case study: Fifteen

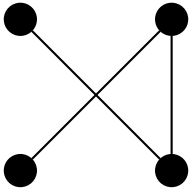
## Trees

A connected graph without any cycles is called a *tree*.

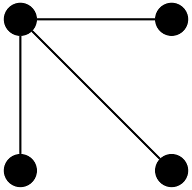
- A tree always has $|V| - 1$ edges.
- There is always a *unique* path between any two vertices in a tree.

A graph (not necessarily connected) without any cycles is called a *forest*. In such a graph, each connected component is a tree.

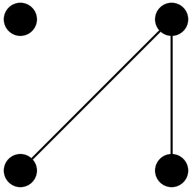
## Tree examples



## Tree examples



## Forest example



## Rooted trees

In a tree we sometimes identify a special vertex, called the *root*. This gives additional organisation to the tree.

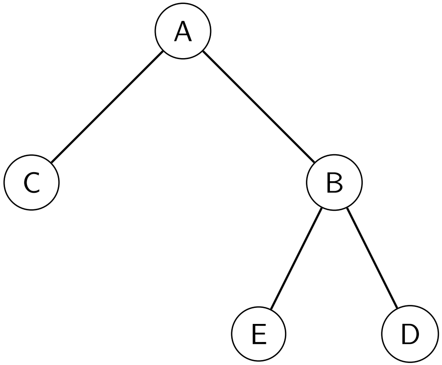
- The *parent* of a vertex is the first vertex in the unique path to the root.
- The *ancestors* of a vertex are its parent, its parent's parent, etc. all the way up to the root.
- The root has no parent.
- Note that the root *must* be specified — it is not intrinsic to the graph. Any vertex can be the root.
- We'll call the parent of $v$ $P(v)$.

## Recursive definition of ancestors

Let's call the set of ancestors of a vertex $v$ $A(v)$. Then

- If $v$ is the root then $A(v) = \emptyset$
- If $v$ is not the root then $A(v) = \{P(v)\} \cup A(P(v))$

## Rooted tree example



The root is chosen to be $A$

- $B$'s ancestors is just $A$
- $D$'s ancestors are $A$ and $B$

## Rooted trees (2)

- The *children* of a vertex are all its neighbours other than its parent.
- The *descendants* of a vertex are all of its children, all of its children's children, etc.
- A vertex without children is called a *leaf*.

## Recursive definition of descendents

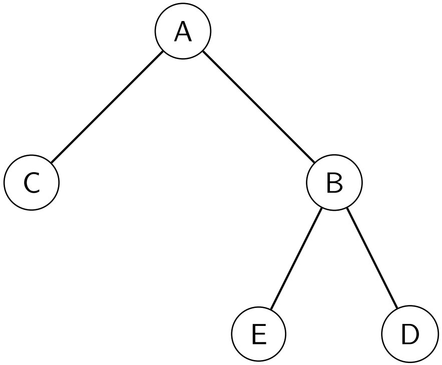
Let's call the set of children of $v$ $C(v)$ and the set of descendents $D(v)$. Then

- If $v$ is the root then $C(v) = N_v$ otherwise $C(v) = N_v \setminus \{P(v)\}$
- The descendents are given by:

$$D(v) = C(v) \cup \bigcup_{u \in C(v)} D(u)$$

- Why is there no base case? If $C(v) = \emptyset$ then the big union will disappear because there will be no $u$ that satisfies.

## Rooted tree example



The root is chosen to be $A$

- $A$'s children are $C$ and $B$.
- $B$'s descendants are $E$ and $D$.
- The leaves are $C, E, D$.

## Sub-graphs

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

- An *induced subgraph* is a subgraph $E'$ which contains *all* edges of $E$ that are incident with vertices in $V'$. We write $G(V')$:

$$G(V') := (V', \{(s, t) \in E : s, t \in V'\})$$

## Sub-trees

If we have a rooted tree, then we can define subtrees.

- A *subtree* of a rooted tree on vertex $v$ is the induced subgraph on $v$ and its descendent
- A subtree is always a rooted tree, and here $v$ is the root.

## Spanning tree

Sometimes we want to identify a tree inside a graph.

- A subgraph $T = (V, E')$ of a connected graph $G = (V, E)$ which is also a tree is called a *spanning tree*.
- If a graph has a spanning tree, then it must be connected, and the spanning tree gives a unique path between any two vertices in $G$.

## Spanning tree uses

- In networking, we often identify a spanning tree of the network and only send packets along the edges of the spanning tree. This prevents packets from going around the network in a cycle.
- The graph of all shortest paths from a vertex will form a rooted spanning tree.
- Many algorithms (A$^*$, Dijkstra's) build a spanning tree as part of their internal structure.
- The cheapest way to distribute power, water, bandwidth, transportation, etc. is to build a spanning tree.

## Outline

Trees

Algorithms for trees

Case study: Teleportation

Case study: Fifteen

## Building spanning trees

We can build spanning trees with BFS or DFS

- Spanning tree from BFS will contain shortest paths from initial vertex to others
- Spanning tree from DFS will not have this property

## Representing spanning trees of shortest paths

How can we easily get a shortest path from root to another vertex from a spanning tree?

- Represent tree as a graph? Still need to get path from tree
- Starting from root, can be many children. Which one to take?
- Each vertex has a unique parent
- Solution: Start from end and follow parents to root
- Unique? Represent parent as a function?

We will use a dictionary of parents to help build the path.

## BFS spanning tree

General idea:

- Use BFS to get distance classes
- To process a vertex, find an arbitrary parent in the previous distance class
- Keep track of parents in a dictionary
- Return the parents dictionary

## BFS spanning trees in Python

```
def arbitrary(S):
   return next(iter(S))

def spanTree(V, E, r):
   parents = { r: None }
   spanTreeR(V - {r}, E, {r}, parents)
   return parents

def spanTreeR(V, E, D, parents):
   Dnew = NS(V, E, D)
   if len(Dnew) == 0: return
   for v in Dnew:
      parents[v] = arbitrary(N(D, E, v))
   spanTreeR(V - Dnew, E, Dnew, parents)
```

## Parents to paths

How to find a path from root to a vertex given the parents?

Go to the vertex's parent, then go to the vertex.

```
def path(parents, v):
   u = parents[v]
   if u == None: return [v]       # at root? Stop
   return path(parents, u) + [v] # go to parent, then to v
```

This returns a list of the vertices in the path from the root to v.

## Outline

Trees

Algorithms for trees

Case study: Teleportation

Case study: Fifteen

## Solving problems with discrete structures

How can we use math to model problems and solve them?

General plan:

- What is the *problem* described in math?
- What does an *instance* of the problem look like?
- How to solve the problem using math?
- What does a solution to an instance look like?
- How to translate the solution back to the original problem?

## Case study: Teleportors

A6 alien race has recently gifted a network of matter transporters to the people of Earth. Matter transporters come in pairs, corresponding to the two ends of a wormhole in space-time, enabling nearly instant transportation between two distance locations. Each of these pairs is referred to as a pont-to-point link. The aliens have installed pairs of transporters in cities around the world, creating a network that allows transportation between any two cities in the network. The aliens have published a list of pairs of cities joined by point-to-point links, like so:

```
Brisbane, Sydney
Vancouver, London
Paris, Brisbane
Paris, Vancouver
Sydney, Paris
```

## Teleportors problem

The aliens charge a fee for using matter transporters. The fee is charged per point-to-point link used, and is the same for all point-to-point links.

A logistics company wishes to send some goods from Brisbane to London and has tasked you with finding the least expensive way of using the matter transportation network to transport these goods, using no other transportation methods.

Let's call this the teleportation transportation problem (TTP)

## The problem, in math

- What are the basic objects in the problem? Cities, point-to-point links.
- How are they related? Each link connects two cities.

This is straightforwardly a graph where the vertices are Cities and the edges are links.

- What do we want to learn? Cheapest path from a city to another city.
- What does cheapest mean? Fewest links.

We want a shortest path between two given cities. (Shortest Path problem: SPP)

## An instance of the problem

To solve a shortest path problem, we need a graph and two vertices. This is an *instance* of the SPP.

We are given a list of links (pairs of cities), a starting city and an end city. This an instance of the TTP.

We will need to translate the TTP into a SPP by deriving the graph and start/end vertices.

## Solving the problem

We've see how to solve the shortest path problem using breadth first seach:

1. Use BFS to traverse the graph from the starting vertex to other vertices, remembering parents.
2. Use parents to derive a graph from the end to the start.

A solution to an instance is just the list of cities in the path, in order. This is straightforwardly the solution that we are looking for. No translation necessary.

## Translating TTP to SPP

First, set up an instance in Python and translate into a graph. We need to figure out the vertices and edges.

```
links = [ ("Brisbane", "Sydney" ), ...]
startCity = "Brisbane"
endCity = "London"
ttpinstance = (links, startCity, endCity) # instance of problem

def ttpToSpp(ttpinstance):
   links, start, end = ttpinstance
   E = links | { (v,u) for (u,v) in links } # symmetrise
   V = { u for (u,v) in E }
   return (V, E, start, end)
```

## Solving in Python

Now we can use functions from before to solve.

```
def solveSpp(V, E, start, end):
   parents = spanTree(V, E, start)
   return path(parents, end)

def solveTTP(instance):
   V, E, start, end = ttpToSpp(instance)
   return solveSpp(V, E, start, end)
```

## Improvements

Our code is not very robust:

- What if start or end city isn't on a link?
- What if the graph is disconnected?
- Code is not very efficient (but is modular!) Could stop BFS once end vertex is found.

Good code will deal with these problems as well!

## Outline

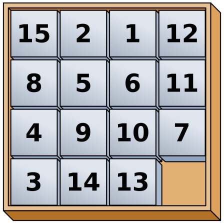
Trees

Algorithms for trees

Case study: Teleportation

Case study: Fifteen

## Fifteen



This classic puzzle asks you to shuffle around tiles to put them into numerical order.

What is the smallest number of moves required? Keep in mind that it might be impossible!

## Nine

We'll approach a similar question for a $3 \times 3$ puzzle.
The Nine problem (9P) is: given an arrangement of the numbers 0 through 8 (0 indicating the empty space), what is the shortest sequence of 15-style moves that obtains the arrangement:

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix}$$

## 9P, in math

This is another shortest path problem! But what is the graph?

- Vertices: arrangements of the numbers 0 through 8
- Edges: between vertices where one is a valid 15-style move from the other

A path in this graph corresponds to a sequence of 15-style moves. A shortest path from the starting configuration to the winning configuration corresponds to an optimal winning sequence of moves.

## 9P instances

An instance of 9P is just the starting configuration, which is an arrangement of the numbers 0 to 8.

List or tuple? There are a fixed number of items, so a tuple makes more sense. And each position has a special meaning, i.e. where the number lives on the board. We'll use tuples.

An argument could be made for using a tuple of rows, each itself a tuple, but this doesn't gain us much.

## Solving the problem

We know how to solve SPP, but where is the graph?

- We could generate the graph and vertices up front, but that would be huge!
- BFS really just needs to know about neighbourhoods
- Instead we'll use a dynamic approach and calculate neighbourhoods on the fly!

## New BFS

Let's fix BFS so that we don't have to rewrite it for every new application!

```
def bfs(D, NS, process, Dold = None):
   if Dold == None: Dold = set()    # base case
   if process(D, Dold): return
   Dnew = NS(D) - D - Dold
   if not Dnew: return
   return bfs(Dnew, NS, process, D)
```

We have separated the traversal logic from the application logic.

## Calculating neighbourhoods

From a vertex, we can calculate the neighbours by the valid moves from that location:

- Move a tile into the space from the left (Move the space right)
- Move a tile into the space from the right (Move the space left)
- Move a tile into the space from above (Move the space up)
- Move a tile into the space from below (Move the space down)

Each of these moves is only valid if there is a tile in the correct position adjacent to the space. Can't move left if the space is already at the left side!

Each type of move is a function from $C$ to $C \cup \{\perp\}$ where $\perp$ means invalid move. We form a neighbourhood by applying each function to a vertex.

## Calculating neighbourhoods in python

```
def swap(puzzle, i, j):
   ret = list(puzzle)
   ret[i] = puzzle[j]
   ret[j] = puzzle[i]
   return tuple(ret)

def moveL(puzzle):
   zeropos = puzzle.index(0)
   if zeropos % 3 == 0: return None # can't move left
   return swap(puzzle, zeropos, zeropos - 1)

# moveR, moveU, moveD similar

moves = { "Left": moveL, "Right": moveR,
   "Up": moveU, "Down": moveD }

def N(puzzle):
   return { move(puzzle) for move in moves.values() } - { None }
```

## Finding the spanning tree

Let's use our new BFS!

```
def spantree(u, N, NS, stopcondition):
    parents = dict()

    def process(D, Dold):
        for v in D:
            vparents = N(v).intersection(Dold)
            parents[v] = arbitrary(vparents) if vparents else None
            if stopcondition(v): return True
        return False

    bfs({u}, NS, process)
    return parents
```

Our `path()` function from before still works to get a path to a vertex from `parents`

## Path to moves

How to get the sequence of moves from the path?

```
def pathToMoves(path):
   def correctMove(p1, p2):
      return arbitrary( { name
         for name, move in moves.items()
         if move(p1) == p2 } )

   return [ correctMove(p1, p2)
      for p1, p2 in zip(path[:-1], path[1:]) ]
```

At each vertex, see which move goes to the next vertex.

## Solving 9p (finally!)

```
solved_puzzle = ( 1, 2, 3,
                  4, 5, 6,
                  7, 8, 0)

def solved(puzzle):
    return puzzle == solved_puzzle

def solve9p(puzzle):
    parents = spantree(puzzle, N, NS, solved)
    thepath = path(parents, solved_puzzle)
    return pathToMoves(thepath)
```

## Improvements

Can we improve this? Yes!

- We use a move to calculate the neighbourhood, but forget it! We later have to figure out the move.
- We move from a parent to a vertex to get the neighbourhood, but forget the parent! Later we have to find a parent.
- The recursion in BFS isn't really doing much. A loop would be fine.

## Improved version

```
def spanningTree2(u, end):
    parents = { u : ( None, None ) }
    D = { u }
    while(D):
        Dnew = set()
        for v in D:
            for name, move in moves.items():
                w = move(v)
                if w == None or w in parents or w in D:
                    continue
                parents[w] = (name, v)
                if w == end: return parents
                Dnew.add(w)
        D = Dnew
    return parents

def path2(parents, v) :
    movename, u = parents[v]
    if u == None: return [ ]
    return path2(parents, u) + [ movename ]
```