# Depth First Search

Depth First Search is a technique for visiting each each vertex of a graph, going as far as possible and then backtracking to visit vertices not yet reached.

We can use depth first search (or breadth first search) to create a spanning tree for a connected graph (a subgraph which is a tree and contains every vertex)



# Depth First Search

As an example we can create a spanning tree T for a connected graph G with vertices $v_1, v_2, \ldots, v_n$.

1. Initialize T to have one vertex, $v_1$, and no edges.
2. visit($v_1$).

Here, visit is a recursive depth first search algorithm.

```
visit(vertex v) {
    mark v visited;
    for each vertex w adjacent to v, not visited {
        add vertex w and edge {v,w} to T;
        visit(w);
        }
    }
```



# Breadth First Search

In breadth first search, instead of going as far as possible, we create a queue to store vertices and visit all the neighbors before moving on.



# Breadth First Search

```
// as an example, to create a spanning tree T.
Let T = tree with only v1, no edges;
Add v1 to queue Q;
While Q is not empty {
    remove v from Q;
    for each neighbor w of v {
      if w not visited {
        add w to Q;
        add w and edge {v,w} to T;
        mark w visited;
      }
    }
}
```



# Applications of Trees

There are numerous important applications of trees, only three of which we will discuss today:

- Network optimization with **minimum spanning trees**
- Problem solving with **backtracking in decision trees**
- Data compression with prefix codes in **Huffman coding trees**



# Spanning Trees

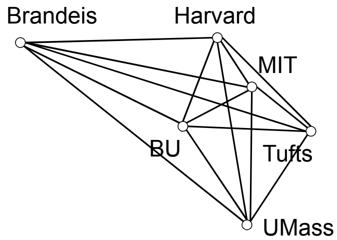
**Definition:** Let G be a connected simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.

**Note:** A spanning tree of G = (V, E) is a connected graph on V with a minimum number of edges (|V| - 1).

**Example:** Since winters in Boston can be very cold, six universities in the Boston area decide to build a tunnel system that connects their libraries.

## Spanning Trees

The complete graph including all possible tunnels:

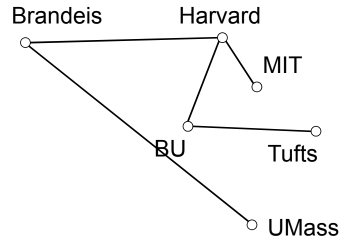



The spanning trees of this graph connect all libraries with a minimum number of tunnels.



## Spanning Trees

Example for a spanning tree:




Since there are 6 libraries, 5 tunnels are sufficient to connect all of them.



## Spanning Trees

Now imagine that you are in charge of the tunnel project. How can you determine a tunnel system of **minimal cost** that connects all libraries?

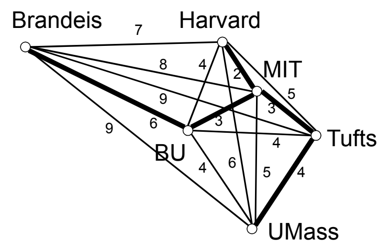
**Definition:** A **minimum spanning tree** in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

How can we find a minimum spanning tree?



## Spanning Trees

The complete graph with cost labels (in billion $):




The least expensive tunnel system costs $18 billion.



## Spanning Trees

**Prim's Algorithm:**

- Begin by choosing any edge with **smallest weight** and putting it into the spanning tree,
- successively add to the tree edges of **minimum weight** that are incident to a vertex already in the tree and not forming a simple circuit with those edges already in the tree,
- stop when (n – 1) edges have been added.



## Spanning Trees

**Kruskal's Algorithm:**

Kruskal's algorithm is identical to Prim's algorithm, except that it does not demand new edges to be incident to a vertex already in the tree.

Both algorithms are **guaranteed** to produce a minimum spanning tree of a connected weighted graph. Kruskal's algorithm is O(e log e) while Prim's algorithm is O(e log v)

Please look at the proof of Prim's algorithm on page 799 (6th edition: p. 741).

## Backtracking in Decision Trees

A **decision tree** is a rooted tree in which each internal vertex corresponds to a decision, with a subtree at these vertices for each possible outcome of the decision.

Decision trees can be used to model problems in which a **series of decisions** leads to a solution (compare with the “binary search tree” example).

The possible **solutions** of the problem correspond to the paths from the root to the leaves of the decision tree.



## Backtracking in Decision Trees

There are problems that require us to perform an **exhaustive search** of all possible sequences of decisions in order to find the solution.

We can solve such problems by constructing the **complete decision tree** and then find a path from its root to a leaf that corresponds to a solution of the problem.

In many cases, the efficiency of this procedure can be dramatically increased by a technique called **backtracking**.



## Backtracking in Decision Trees

**Idea:** Start at the root of the decision tree and move downwards, that is, **make a sequence of decisions**, until you either reach a solution or you enter a situation from where no solution can be reached by any further sequence of decisions.

In the latter case, **backtrack to the parent** of the current vertex and take a different path downwards from there. If all paths from this vertex have already been explored, backtrack to its parent.

Continue this procedure until you **find a solution** or establish that **no solution exists** (there are no more paths to try out).



## Backtracking in Decision Trees

**Example:** The n-queens problem

How can we place n queens on an n×n chessboard so that no two queens can capture each other?

A queen can move any number of squares horizontally, vertically, and diagonally.

Here, the possible target squares of the queen Q are marked with an x.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| x | | | x | | | x | |
| | x | | x | | x | | |
| | | x | x | x | | | |
| x | x | x | Q | x | x | x | x |
| | | x | x | x | | | |
| | x | | x | | x | | |
| x | | | x | | | x | |
| | | | x | | | | x |



## Backtracking in Decision Trees

Obviously, in any solution of the n-queens problem, there must be **exactly one queen in each column** of the board.

Therefore, we can describe the solution of this problem as a **sequence of n decisions**:

Decision 1: Place a queen in the first column.
Decision 2: Place a queen in the second column.
:
:
Decision n: Place a queen in the n-th column.

We are now going to solve the **4-queens problem** using the backtracking method.



## Backtracking in Decision Trees

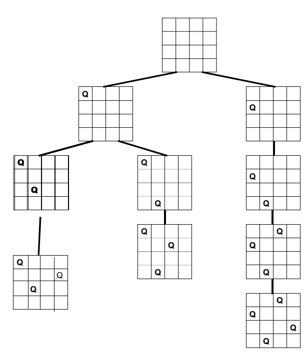
empty board

place 1st queen

place 2nd queen

place 3rd queen

place 4th queen



## Backtracking in Decision Trees

We can also use backtracking to write “intelligent” programs that play games against a human opponent.

Just consider this extremely simple (and not very exciting) game:

At the beginning of the game, there are seven coins on a table. Player 1 makes the first move, then player 2, then player 1 again, and so on. One move consists of removing 1, 2, or 3 coins. The player who removes all remaining coins wins.



## Backtracking in Decision Trees

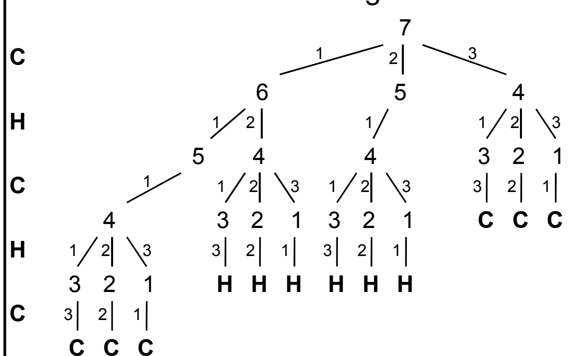
Let us assume that the computer has the first move. Then, the game can be described as a **series of decisions**, where the first decision is made by the computer, the second one by the human, the third one by the computer, and so on, until all coins are gone.

The **computer** wants to make decisions that **guarantee its victory** (in this simple game).

The underlying assumption is that the **human** always finds the **optimal move**.



## Backtracking






## Backtracking in Decision Trees

So the computer will start the game by taking three coins and is **guaranteed to win** the game.

For more interesting games such as **chess**, it is **impossible** to check every possible sequence of moves. The computer player then only looks ahead a certain number of moves and **estimates** the chance of winning after each possible sequence.



## Permutation Matrices

A permutation matrix is an n by n matrix with a single 1 in each row and column, 0 elsewhere.

If P is a permutation (bijection) on {1,2,..,n} let $A_P$ be the permutation matrix with $A_{iP(i)} = 1$, $A_{ij} = 0$ for $j \neq P(i)$



## Permutation Matrices

Let $E_{uv}$ be the n by n matrix with 1 in the (u,v) position and 0 elsewhere.

* Note that $E_{uv}E_{rs} = E_{us}$ if v = r, and is the n by n zero matrix otherwise.

Then $A_P = \Sigma_{i=1}^{n} E_{iP(i)}$

If $Q = P^{-1}$ then you can check that $A_Q = (A_P)^T$, the transpose of $A_P$.

## Permutation Matrices

Note also that

$A_P^T A_P = \Sigma_{i=1}^n E_{P(i)i} \Sigma_{t=1}^n E_{tP(t)} = \Sigma_{i=1}^n \Sigma_{t=1}^n E_{P(i)i} E_{tP(t)} = \Sigma_{t=1}^n E_{P(t)P(t)} = I_n$, the n by n identity matrix, by *

Also, $E_{ik} A_P = E_{ik} \Sigma_{t=1}^n E_{tP(t)} = E_{iP(k)}$.

Right multiplying an m by n matrix B by $A_P$ permutes the columns of B, moving the $k^{th}$ column to the $P(k)^{th}$ column



## Permutation Matrices

Likewise $A_P^T E_{ik} = \Sigma_{t=1}^n E_{P(t)t} E_{ik} = E_{P(i)k}$ so left multiplying an n by m matrix B by $A_P^T$ will permute the rows, moving the $i^{th}$ row to the $P(i)^{th}$ place.

If B is n by n then $A_P^T B A_P$ will be B with both rows and columns permuted by P: row i $\rightarrow$ row P(i), column j $\rightarrow$ column P(j)



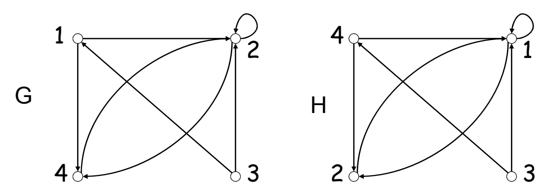
## Isomorphisms of Graphs

Suppose G and H are graphs, each with n vertices. If G has vertices $g_1, \ldots g_n$ and H has vertices $h_1, \ldots, h_n$ then a permutation P taking $g_i$ to $h_{P(i)}$ will give an isomorphism of graphs if $A_P^T M_G A_P = M_H$, where $M_S$ is the adjacency matrix of graph S.



### Example

The graphs G and H are clearly isomorphic, but can we tell that from their matrices?




Map vertices of G to those of H by P(1) = 4, P(2) = 1, P(3) = 3, P(4) = 2.



$$M_G = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \qquad M_H = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$A_P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$



$$M_G A_P = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$A_P^T M_G A_P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = M_H$$

## Isomorphisms of Graphs

Note that if F is an isomorphism from a graph G of n vertices $v_1 \ldots v_n$ to a graph H of n vertices $w_1 \ldots w_n$ then F defines a permutation of {1,…,n} and the adjacency matrices of G and H will be related by a permutation matrix.



## Isomorphisms of Graphs

But not every permutation of the vertices will produce a graph isomorphism. The permutations producing a graph isomorphism F have to map the edges appropriately because (v,u) is an edge iff (F(v), F(u)) is an edge.