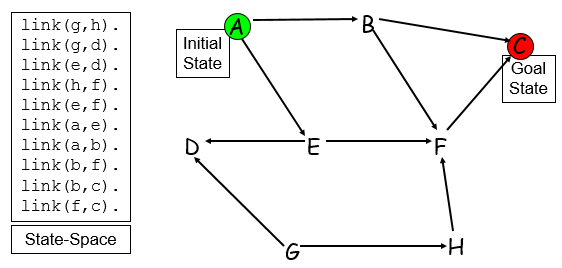
**Lecture 2**

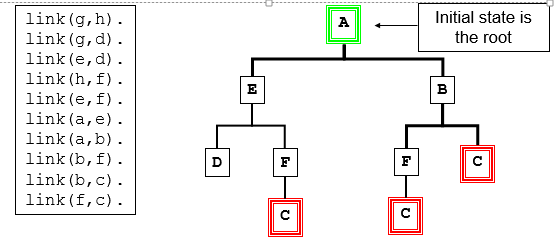
**State-Space Search**

* Many problems in AI take the form of ***state-space search***.
* The ***states*** might be legal board configurations in a game, towns and cities in some sort of route map, collections of mathematical propositions, etc.
* The ***state-space*** is the configuration of the possible states and how they connect to each other e.g. the legal moves between states.
* When we don't have an ***algorithm*** which tells us definitively how to negotiate the state-space we need to search the state-space to find an optimal path from a start state to a goal state.
* We can only decide what to do (or where to go), by considering the possible moves from the current state, and trying to look ahead as far as possible. Chess, for example, is a very difficult state-space search problem.

**2.1 An example problem: Searching a graph**



**2.2 State-Space Representation**

****

* An abstract representation of a state-space is a downwards growing tree. Connected *nodes* represent states in the domain.
* The *branching factor* denotes how many new states you can move to from any state. This problem has an average of 2.
* The *depth* of a node denotes how many moves away from the initial state it is. ‘C’ has two depths, 2 or 3.

**2.3 Searching for the optimum**

* State-space search is all about finding, in a state-space (which may be ***extremely*** large: e.g. chess), some *optimal state/node*.
* `Optimal' can mean very different things depending on the nature of the domain being searched.
* For a puzzle, `optimal' might mean the goal state e.g. connect4
* For a route-finder, like our problem, which searches for shortest routes between towns, or components of an integrated circuit, `optimal' might mean the shortest path between two towns/components.
* For a game such as chess, in which we typically can't see the goal state, `optimal' might mean the best move we think we can make, *looking ahead* to see what effects the possible moves have.

**2.4 Implementing**

To implement state-space search in Prolog, we need:

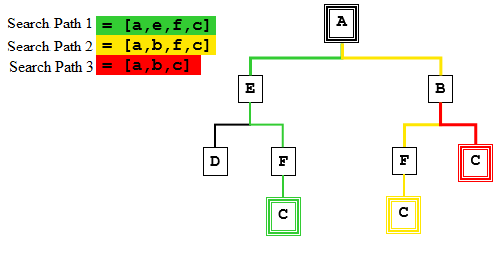
1. A way of representing a state e.g. the board configuration
   * Link (a , e).
2. A way of generating all of the next states reachable from a given state;
   * Go ( X , Y ) :- link ( X , Y ).

3. A way of determining whether a given state is the one we're looking for. Sometimes this might be the goal state (a finished puzzle, a completed route, a checkmate position); other times it might simply be the state we estimate is the best, using some evaluation function;

4. A mechanism for controlling how we search the space.

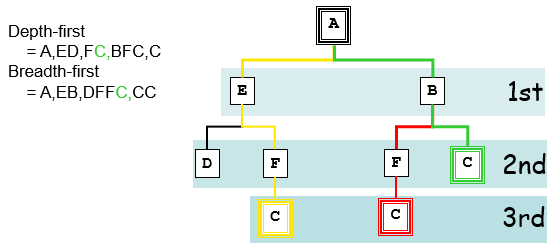
**2.5 Depth-First Search**

* This simple search algorithm uses Prolog’s unification routine to find the first link from the current node then follows it.
* It always follows the left-most branch of the search tree first; following it down until it either finds the goal state or hits a dead-end. It will then backtrack to find another branch to follow.

****

**2.6 Breadth-First Search**

* A simple, common alternative to depth-first search is: ***breadth-first search***.
* This checks every node at one level of the space, before moving onto the next level.
* It is distinct from iterative deepening as it maintains a list of alternative candidate nodes that can be expanded at each depth

****

**2.7 Depth-first vs. Breadth-first**

Advantages of breadth-first:

* Guaranteed to find a solution (if one exists);
* Depending on the problem, can be guaranteed to find an *optimal* solution.

Disadvantages of breadth-first:

* More complex to implement;
* Needs a lot of memory for storing the state space if the search space has a high branching factor.

Advantages of depth-first:

* Simple to implement;
* Needs relatively small memory for storing the state-space.

Disadvantages of depth-first:

* Can sometimes fail to find a solution;
* Not guaranteed to find an *optimal* solution;
* Can take a lot longer to find a solution.