6.034 Artificial Intelligence. Copyright © 2004 by Massachusetts Institute of Technology.

6.034 Notes: Section 2.1

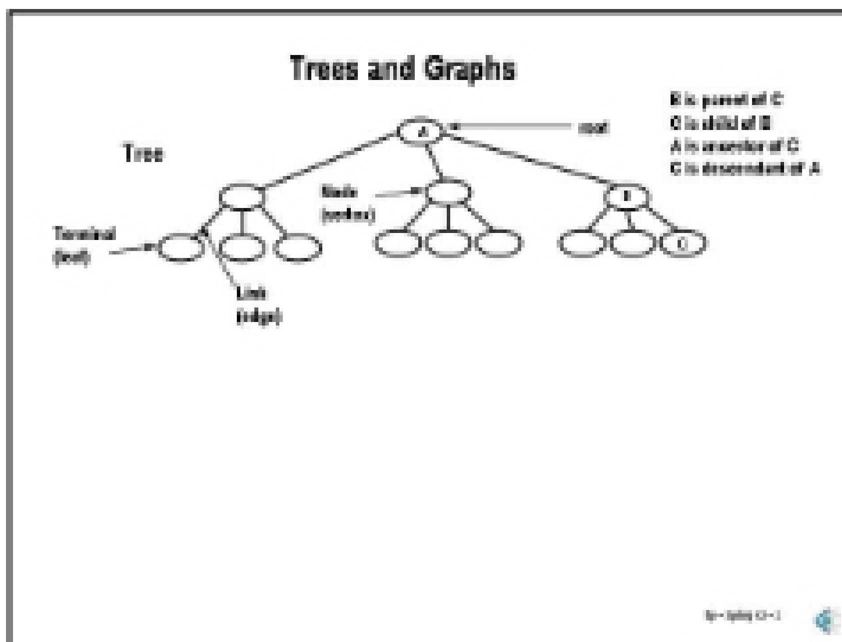
Slide 2.1.1

Search plays a key role in many parts of AI. These algorithms provide the conceptual backbone of almost every approach to the systematic exploration of alternatives.

We will start with some background, terminology and basic implementation strategies and then cover four classes of search algorithms, which differ along two dimensions: First, is the difference between **uninformed** (also known as **blind**) search and then **informed** (also known as **heuristic**) searches. Informed searches have access to task-specific information that can be used to make the search process more efficient. The other difference is between **any path** searches and **optimal** searches. Optimal searches are looking for the best possible path while any-path searches will just settle for finding some solution.

6.034 Artificial Intelligence

- Big idea: Search allows exploring alternatives
- Background
- Uninformed vs Informed
- Any Path vs Optimal Path
- Implementation and Performance



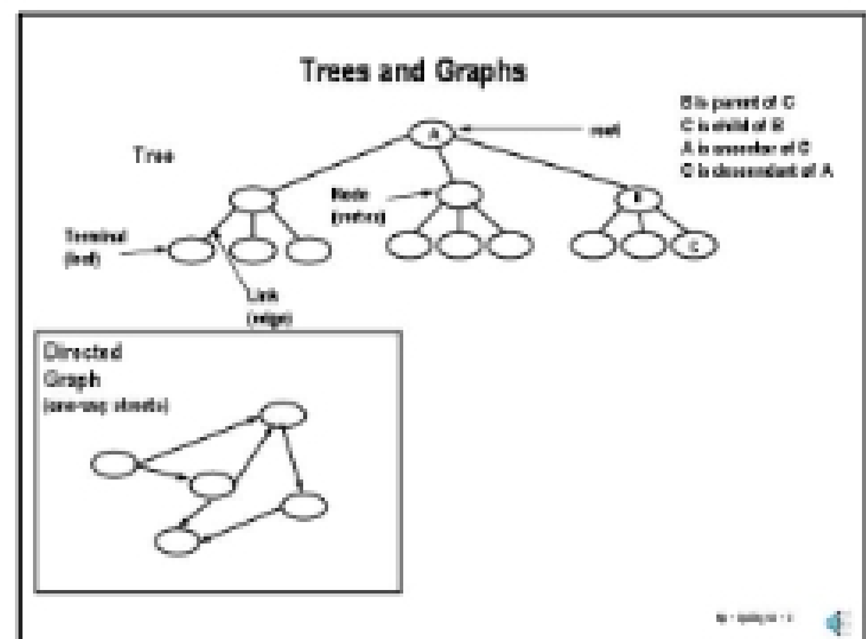
Slide 2.1.2

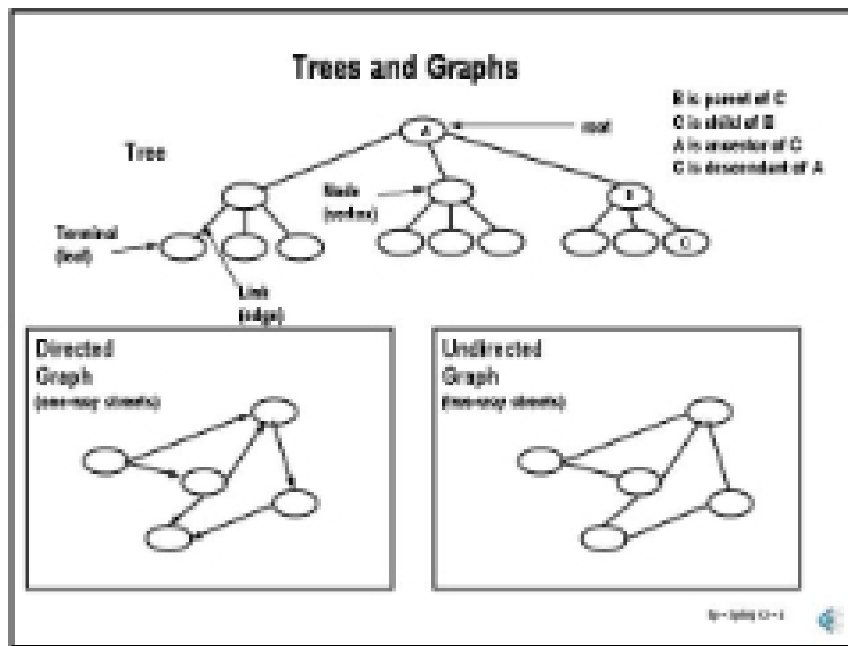
The search methods we will be dealing with are defined on trees and graphs, so we need to fix on some terminology for these structures:

- A tree is made up of **nodes** and **links** (circles and lines) connected so that there are no loops (cycles). Nodes are sometimes referred to as vertices and links as edges (this is more common in talking about graphs).
- A tree has a **root** node (where the tree "starts"). Every node except the root has a single **parent** (aka **direct ancestor**). More generally, an **ancestor** node is a node that can be reached by repeatedly going to a parent node. Each node (except the **terminal** (aka **leaf**) nodes) has one or more **children** (aka **direct descendants**). More generally, a **descendant** node is a node that can be reached by repeatedly going to a child node.

Slide 2.1.3

A graph is also a set of nodes connected by links but where loops are allowed and a node can have multiple parents. We have two kinds of graphs to deal with: **directed** graphs, where the links have direction (akin to one-way streets).



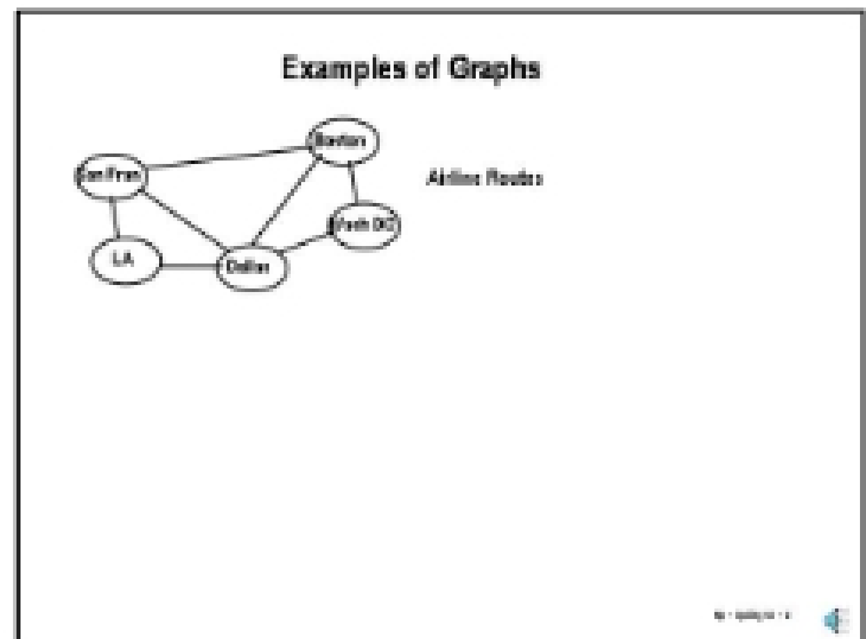


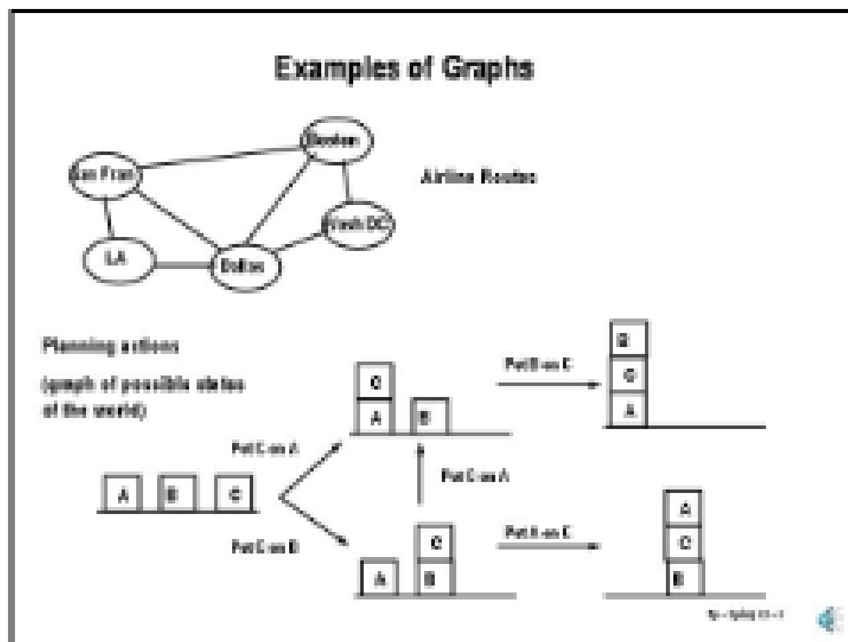
Slide 2.1.4

And, **undirected** graphs where the links go both ways. You can think of an undirected graph as shorthand for a graph with directed links going each way between connected nodes.

Slide 2.1.5

Graphs are everywhere; for example, think about road networks or airline routes or computer networks. In all of these cases we might be interested in finding a path through the graph that satisfies some property. It may be that any path will do or we may be interested in a path having the fewest "hops" or a least cost path assuming the hops are not all equivalent, etc.





Slide 2.1.6

However, graphs can also be much more abstract. Think of the graph defined as follows: the nodes denote descriptions of a state of the world, e.g., which blocks are on top of what in a blocks scene, and where the links represent actions that change from one state to the other.

A path through such a graph (from a start node to a goal node) is a "plan of action" to achieve some desired goal state from some known starting state. It is this type of graph that is of more general interest in AI.

Slide 2.1.7

One general approach to problem solving in AI is to reduce the problem to be solved to one of searching a graph. To use this approach, we must specify what are the **states**, the **actions** and the **goal test**.

A state is supposed to be **complete**, that is, to represent all (and preferably only) the relevant aspects of the problem to be solved. So, for example, when we are planning the cheapest round-the-world flight plan, we don't need to know the address of the airports; knowing the identity of the airport is enough. The address will be important, however, when planning how to get from the hotel to the airport. Note that, in general, to plan an air route we need to know the airport, not just the city, since some cities have multiple airports.

We are assuming that the actions are **deterministic**, that is, we know exactly the state after the action is performed. We also assume that the actions are **discrete**, so we don't have to represent what happens while the action is happening. For example, we assume that a flight gets us to the scheduled destination and that what happens during the flight does not matter (at least when planning the route).

Problem Solving Paradigm

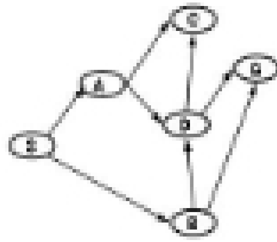
- What are the **states**? (All relevant aspects of the problem)
  - Arrangement of parts (to plan an assembly)
  - Positions of trucks (to plan package distribution)
  - City (to plan a trip)
  - Set of facts (e.g. to prove geometry theorem)
- What are the **actions** (operators)? (Deterministic and discrete)
  - Assemble two parts
  - Move a truck to a new position
  - Fly to a new city
  - Apply a theorem to derive new fact
- What is the **goal test**? (Conditions for success)
  - All parts in place
  - All packages delivered
  - Reached destination city
  - Derived goal fact

Note that we've indicated that (in general) we need a test for the goal, not just one specific goal state. So, for example, we might be interested in any city in Germany rather than specifically Frankfurt. Or, when proving a theorem, all we care is about knowing one fact in our current data base of facts. Any final set of facts that contains the desired fact is a proof.

In principle, we could also have multiple starting states, for example, if we have some uncertainty about the starting state. But, for now, we are not addressing issues of uncertainty either in the starting state or in the result of the actions.

Graph Search as Tree Search

- Trees are directed graphs without cycles and with nodes having <= 1 parent
- We can turn graph search problems into tree search problems by:
  - replacing undirected links by 2 directed links
  - avoiding loops in path (or keeping track of visited nodes globally)



Slide 2.1.8

Note that trees are a subclass of directed graphs (even when not shown with arrows on the links). Trees don't have cycles and every node has a single parent (or is the root). Cycles are bad for searching, since, obviously, you don't want to go round and round getting nowhere.

When asked to search a graph, we can construct an equivalent problem of searching a tree by doing two things: turning undirected links into two directed links; and, more importantly, making sure we never consider a path with a loop or, even better, by never visiting the same node twice.

Slide 2.1.9

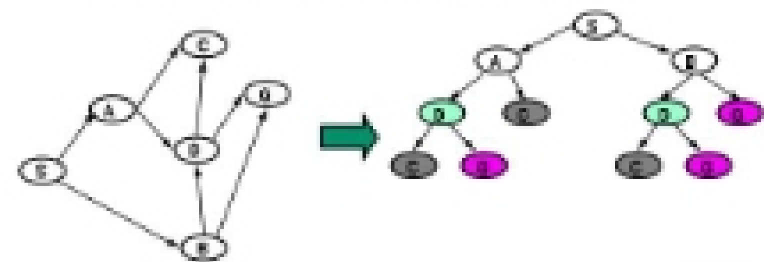
You can see an example of this converting from a graph to a tree here. If we assume that S is the start of our search and we are trying to find a path to G, then we can walk through the graph and make connections from every node to every connected node that would not create a cycle (and stop whenever we hit G). Note that such a tree has a leaf node for every non-looping path in the graph starting at S.

Also note, however, that even though we avoided loops, some nodes (the colored ones) are duplicated in the tree, that is, they were reached along different non-looping paths. This means that a complete search of this tree might do extra work.

The issue of how much effort to place in avoiding loops and avoiding extra visits to nodes is an important one that we will revisit later when we discuss the various search algorithms.

Graph Search as Tree Search

- Trees are directed graphs without cycles and with nodes having <= 1 parent
- We can turn graph search problems (from S to G) into tree search problems by:
  - replacing undirected links by 2 directed links
  - avoiding loops in path (or keeping track of visited nodes globally)



Terminology

- **State** – Used to refer to the vertices of the underlying **graph** that is being searched, that is, states in the problem domain, for example, a city, an arrangement of blocks or the arrangement of parts in a puzzle.
- **Search Node** – Refers to the vertices of the search **tree** which is being generated by the search algorithm. Each node refers to a state of the world; **many nodes may refer to the same state**. Importantly, a node implicitly represents a path (from the start state of the search to the state associated with the node). Because search nodes are part of a search tree, they have a unique ancestor node (except for the root node).

Slide 2.1.10

One important distinction that will help us keep things straight is that between a **state** and a **search node**.

A state is an arrangement of the real world (or at least our model of it). We assume that there is an underlying "real" state graph that we are searching (although it might not be explicitly represented in the computer; it may be implicitly defined by the actions). We assume that you can arrive at the same real world state by multiple routes, that is, by different sequences of actions.

A search node, on the other hand, is a data structure in the search algorithm, which constructs an explicit tree of nodes while searching. Each node refers to some state, but not uniquely. Note that a node also corresponds to a path from the start state to the state associated with the node. This follows from the fact that the search algorithm is generating a **tree**. So, if we return a node, we're returning a path.