UCS1524 – Logic Programming

Problem Solving Strategies –
DFS and BFS

Session Meta Data

Author	Dr. D. Thenmozhi
Reviewer	
Version Number	1.2
Release Date	20 October 2022

Session Objectives

- Understanding problem solving strategies in Prolog.
- Learn about depth first search (DFS) and breadth first search (BFS) algorithms.

Session Outcomes

- At the end of this session, participants will be able to
  - Apply the DFS and BFS for problem solving in Prolog.

Agenda

- Problem solving strategies
  - Problem formulation
  - State –space
  - Search problem
- DFS
  - Algorithm
  - Algorithm for closed set
- BFS

Problem formulation

- a *search problem* is defined in terms of states, operators and goals
- a **state** is a complete description of the world for the purposes of problem-solving
  - the **initial state** is the state the world is in when problem solving begins
  - a **goal state** is a state in which the problem is solved
- an **operator** is an action that transforms one state of the world into another state

Goal states

- depending on the number of solutions a problem has, there may be a single goal state or many goal states:
  - in the eight-puzzle there is a single correct configuration of tiles and a single goal state
  - in chess, there are many winning positions, and hence many goal states
- to avoid having to list all the goal states, the goal is often specified implicitly in terms of a *test* on states which returns true if the problem is solved in a state

Applicable operators

- in general, not all operators can be applied in all states
  - in a given chess position, only some moves are legal (as defined by the rules of chess)
  - in a given eight-puzzle configuration, only some moves are physically possible
- the set of operators which are *applicable* in a state *s* determine the states that can be reached from *s*

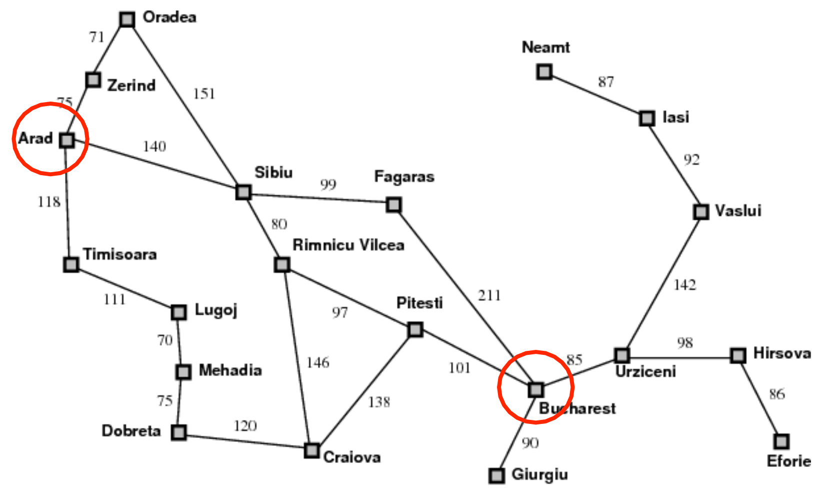
Example: route planning

- **states:** ‘being in *X*’, where *X* is one of the cities

- **initial state**: being in Arad

- **goal state**: being in Bucharest

- **operators**: actions of driving from city X to city Y along the connecting roads, e.g, driving from Arad to Sibiu



State space

- the initial state and set of operators together define the *state space* – the set of all states reachable from the initial state by any sequence of actions

- a *path* in the state space is any sequence of actions leading from one state to another

- even if the number of states is finite, the number of paths may be infinite, e.g., if it possible to reach state B from state A and vice versa

Definition of search problem

- a *search problem* is defined by:
  - a *state space* (i.e., an initial state or set of initial states and a set of operators)
  - a *set of goal states* (listed explicitly or given implicitly by means of a property that can be applied to a state to determine if it is a goal state)
- a *solution* is a path in the state space from an initial state to a goal state

Goals vs solutions

- the *goal* is what we want to achieve, e.g.,
  - a particular arrangement of tiles in the eight-puzzle, being in a particular city in the route planning problem, winning a game of chess, etc.
- a *solution* is a sequence of actions (operator applications) that achieve the goal, e.g,
  - how the tiles should be moved in the eight-puzzle, which route to take in the route planning problem, which moves to make in chess etc.

Example state space

Route Planning Problem

- **states**: ‘being in *X*’, where *X* is one of the cities
- **initial state**: being in *Arad*
- **goal state**: being in *Bucharest*
- **operators**: driving from city *X* to city *Y* along the connecting roads, e.g, driving from *Arad* to *Sibiu*

- **state space**: is the set of all cities reachable from *Arad*

- **solution**: if we place no constraints on the length of the route, any path from *Arad* to *Bucharest* is a solution

Exploring the state space

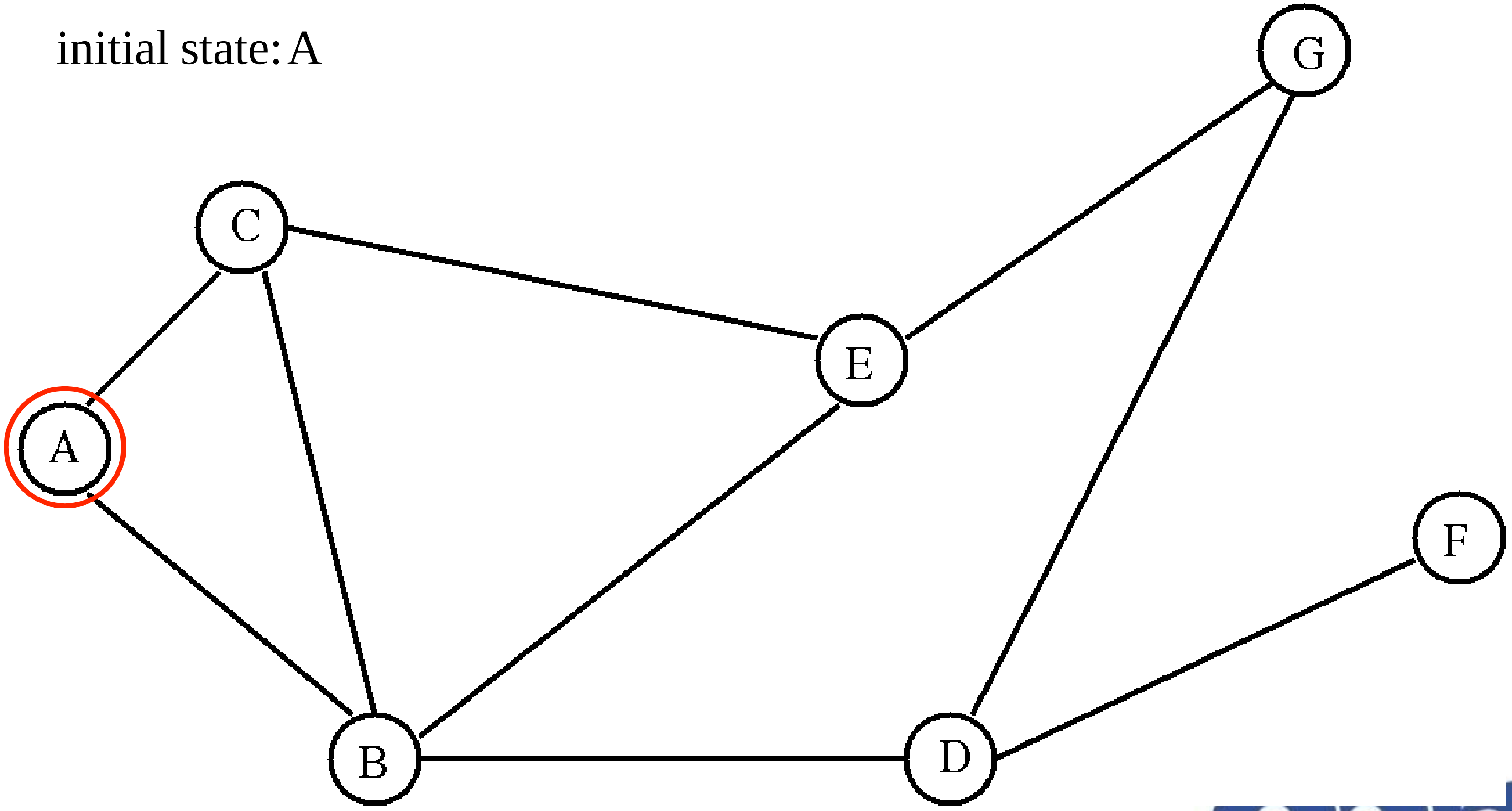
- *search* is the process of exploring the state space to find a solution
- exploration starts from the initial state
- the search procedure applies operators to the initial state to generate one or more new states which are hopefully nearer to a solution
- the search procedure is then applied recursively to the newly generated states
- the procedure terminates when either a solution is found, or no operators can be applied to any of the current states

Search trees

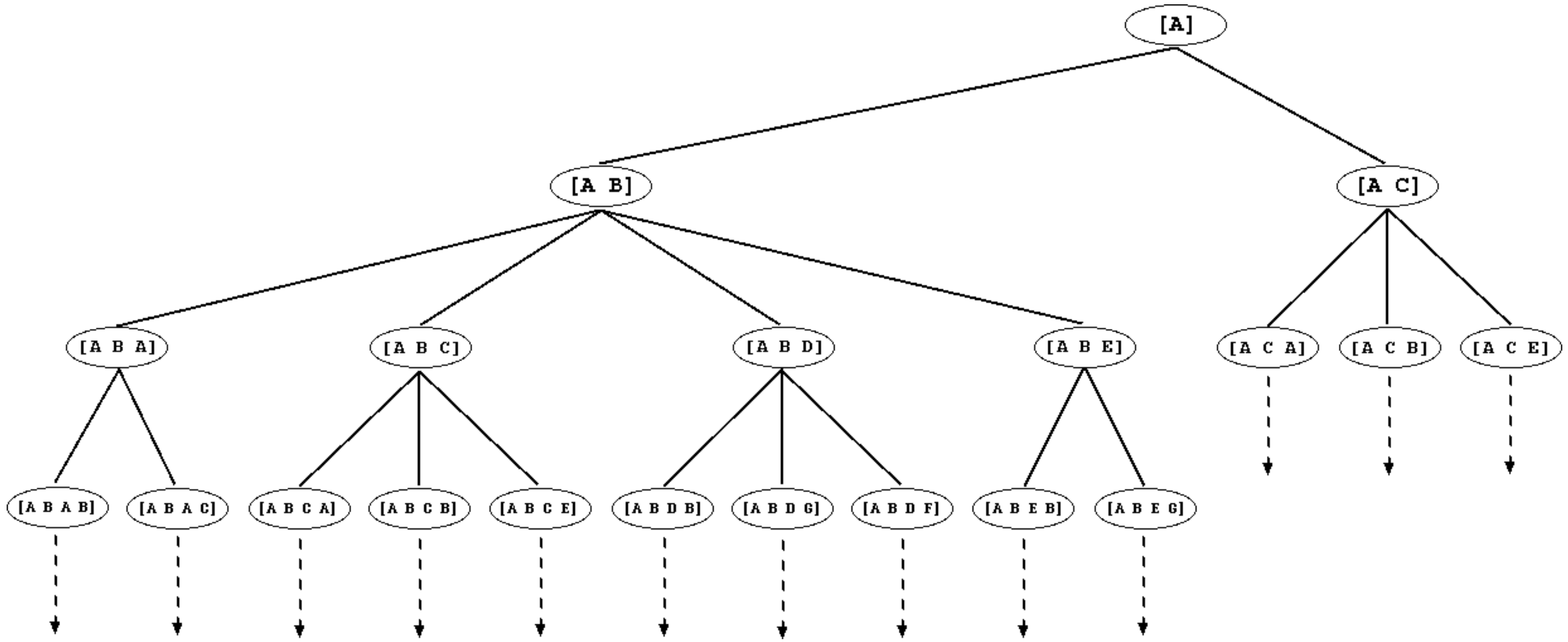
- the part of the state space that has been explored by a search procedure can be represented as a *search tree*
- nodes in the search tree represent *paths* from the initial state (i.e., partial solutions) and edges represent operator applications
- the process of generating the children of a node by applying operators is called *expanding* the node
- the branching factor of a search tree is the average number of children of each non-leaf node
- if the branching factor is b, the number of nodes at depth d is b^d

Example: state space

initial state: A



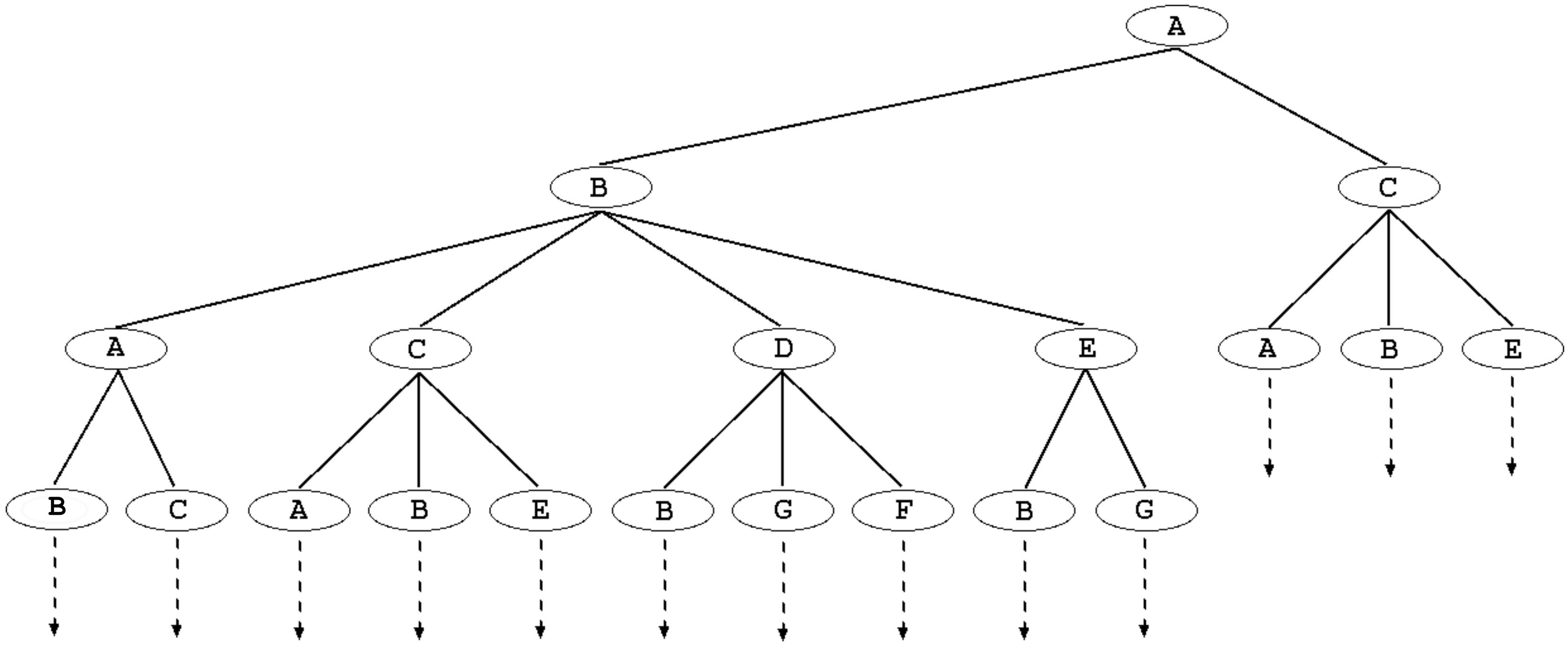
Example: search tree



States vs nodes

- *states* in the state space represent *states of the world*
- *nodes* in the search tree are data structures maintained by a search procedure representing *paths to a particular state*
- the same state can appear in several nodes if there is more than one path to that state
- the nodes of a search tree are often labelled with only the name of the *last state* on the corresponding path
- the path can be reconstructed by following edges back to the root of the tree

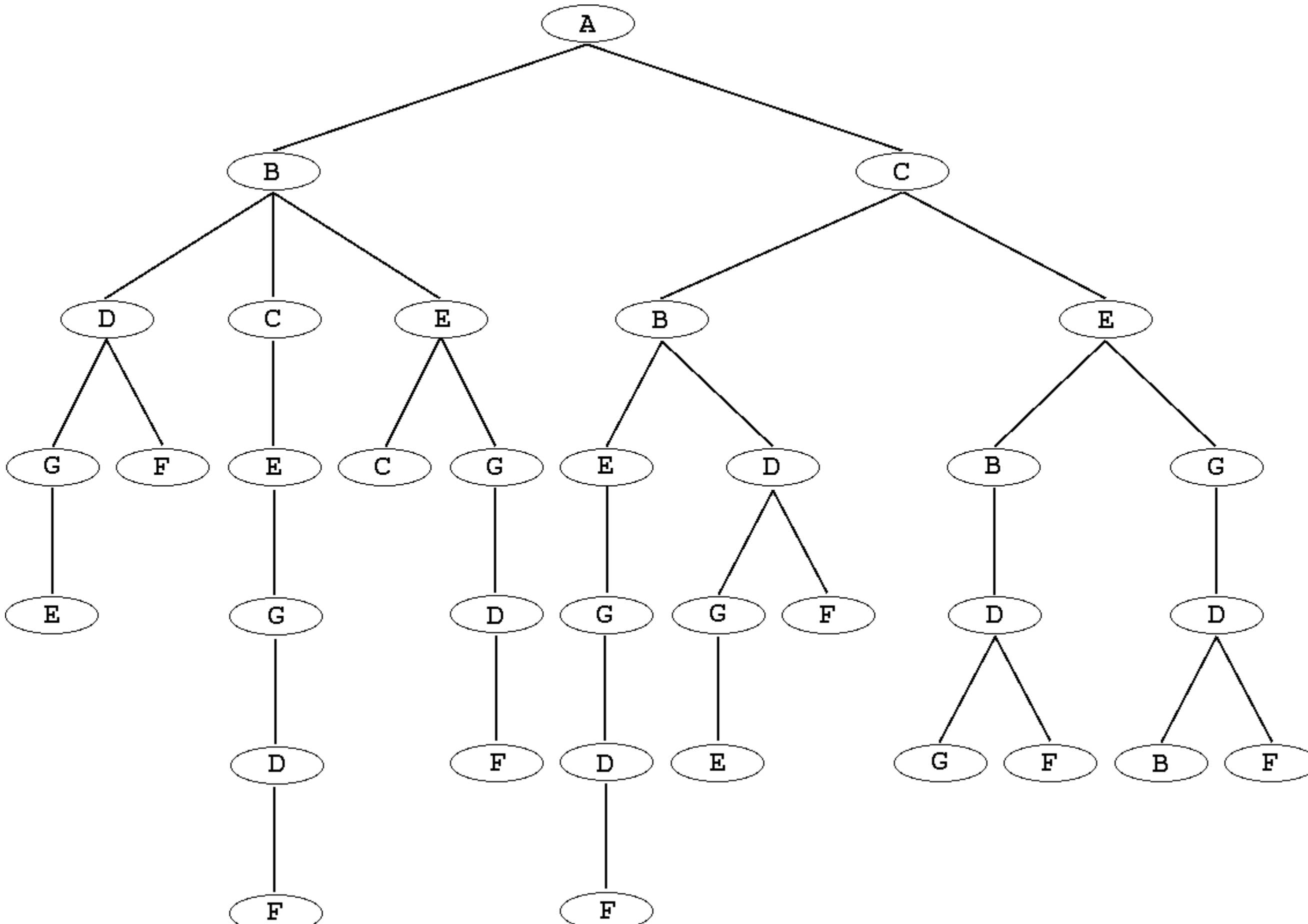
Example: labelling nodes



Eliminating loops

- *paths* containing loops take us back to the same state and so can contribute nothing to the solution of the problem
- e.g., the path *A*, *B*, *A*, *B* is a valid path from *A* to *B* but does not get us any closer to, say *F*, than the path *A*, *B*
- for some problems, e.g., the route planning problem, eliminating loops transforms an *infinite search tree* into a *finite search tree*
- however eliminating loops can be *computationally expensive*

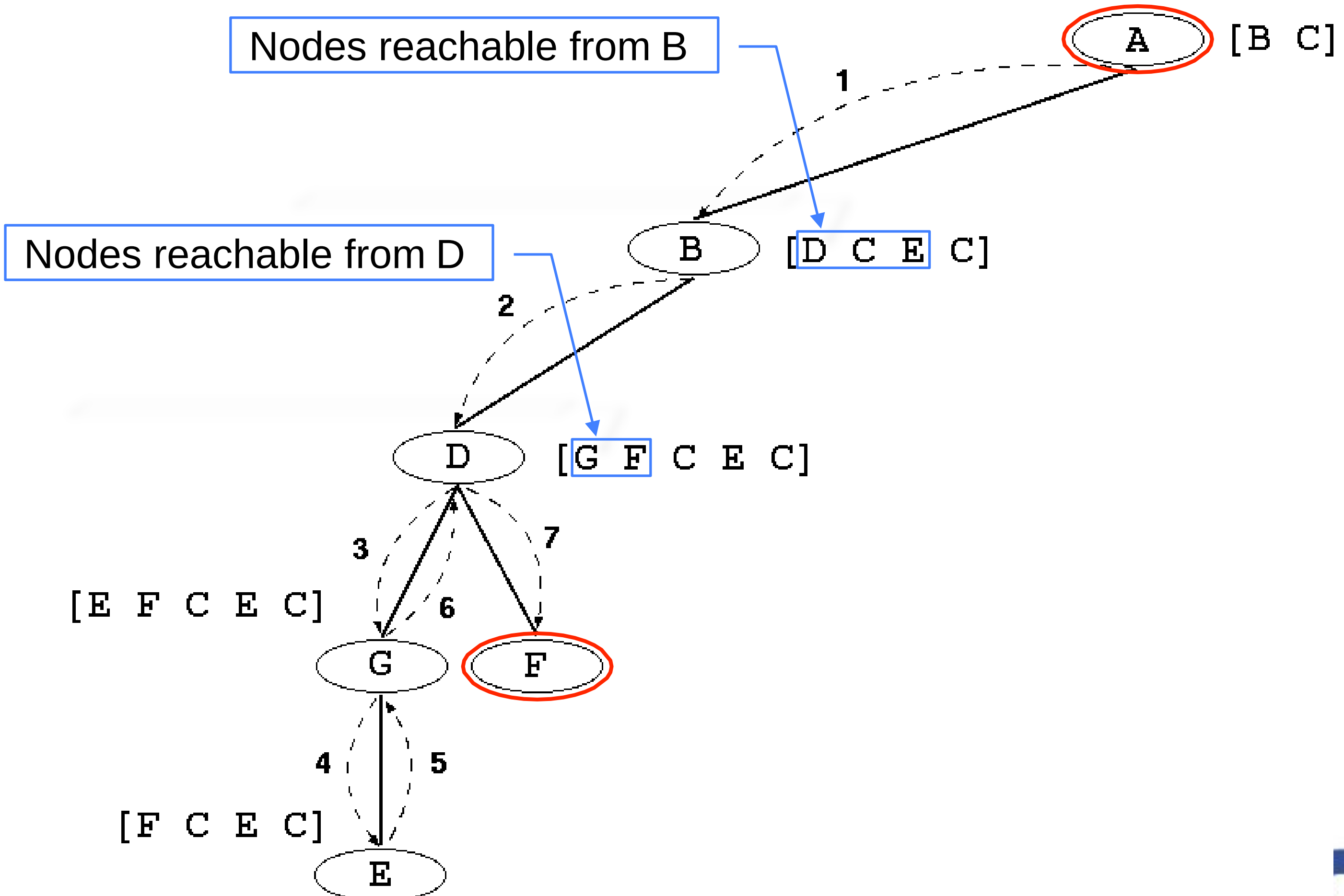
Example: eliminating loops



Depth-first search

- proceeds down a single branch of the tree at a time
- expands the root node, then the leftmost child of the root node, then the leftmost child of that node etc.
- always expands a node at the deepest level of the tree
- only when the search hits a dead end (a partial solution which can't be extended) does the search *backtrack* and expand nodes at higher levels

Example: depth-first search



(Depth first) search in Prolog

- to implement search in Prolog we need to decide ...

- how to represent the search problem:

  – states – what properties of states are relevant
  – operators – including the applicability of operators
  – goals – should these be represented as a set of states or a test

- how to represent the search tree and the state of the search:

  – paths – what information about a path is relevant (cost(s) etc)
  – nodes – parent, children, depth in the tree etc.
  – open/closed lists

DFS

To find a solution path, **Sol**, from a given node, **N**, to some goal node:

- if N is a goal node then **Sol** = [N], or
- if there is a successor node, N1, of N, such that there is a path Sol1 from N1 to a goal node, then **Sol** = [N | Sol1].

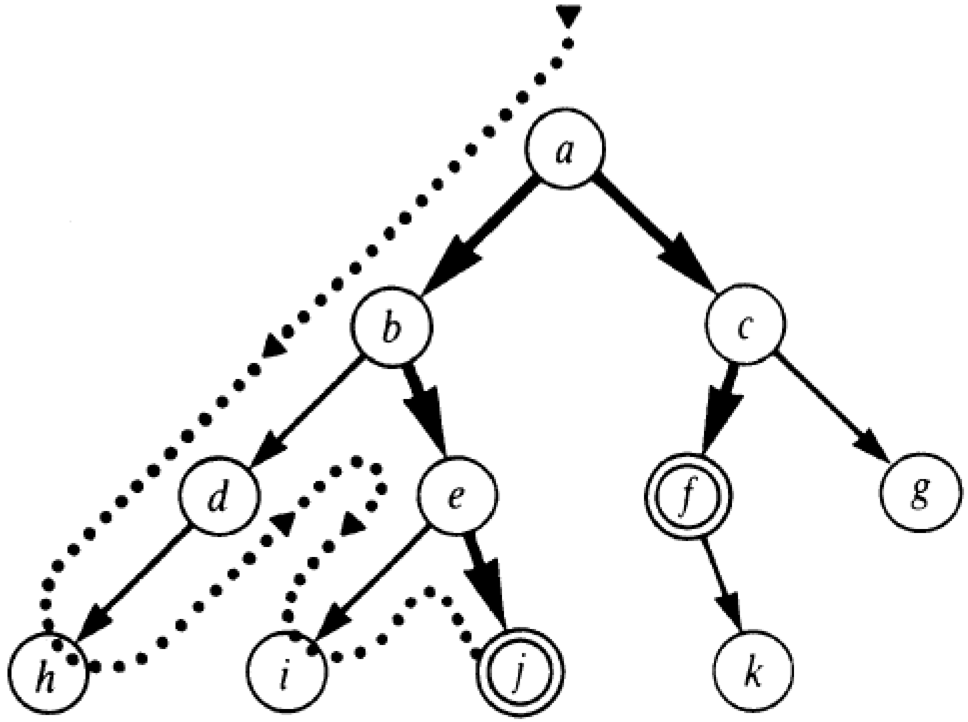
- solve(N, [N]) :- goal(N).
- solve(N, [N | Sol1]) :- s(N, N1), solve(N1, Sol1).

DFS

```
% solve( Node, Solution):
% Solution is an acyclic path (in reverse order) between
    Node and a goal
solve( Node, Solution) :- depthfirst( [], Node, Solution).
% depthfirst( Path, Node, Solution):
% extending the path [Node | Path] to a goal gives Solution
depthfirst( Path, Node, [Node | Path] ) :- goal( Node).
depthfirst( Path, Node, Sol) :- s( Node, Node1),
                    \+ member( Node1, Path), % Prevent a cycle
                   depthfirst( [Node | Path], Node1, Sol).
```

DFS Example

goal(f).
goal(j).
s(a,b).
s(a,c).
s(b,d).
s(b,e).
s(c,f).
s(c,g).
s(d,h).
s(e,i).
s(e,j).
s(f,k).



Solution: [a, b, e, j]   in reverse
[a, c, f]

BFS

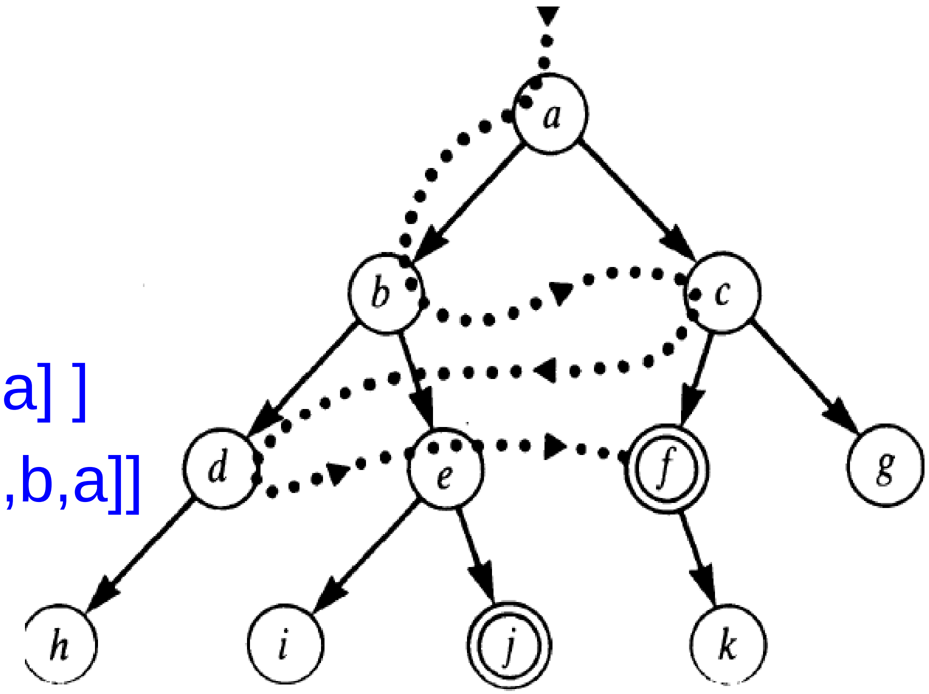
To do the breadth-first search when given a set of candidate paths:

- if the first path contains a goal node as its head then this is a solution of the problem, otherwise
- remove the first path from the candidate set and generate the set of all possible one-step extensions of this path, adding this set of extensions at the end of the candidate set, and execute breadth-first search on this updated set.

- Start with initial candidate set (e.g. [[a]])
- Generate extensions of [a] ([[b,a], [c,a]] : inverse order)
- Remove first candidate [b,a], generate extension and add to end of the candidate set
- [[c, a],[d, b, a], [e, b, a]]

BFS - Example

- Goal : [f, j]
- [[a]]
- [[b,a], [c,a]]
- [[c,a], [d,b,a], [e,b,a]]
- [[d,b,a], [e,b,a], [f,c,a], [g,c,a]]
- [[e,b,a], [f,c,a], [g,c,a] ,[h,d,b,a]]
- [[f,c,a],[g,c,a],[h,d,b,a],
  [i,e,b,a],[j,e,b,a]
- [f,c,a] contains a goal node, f
  returns the solution



BFS Code

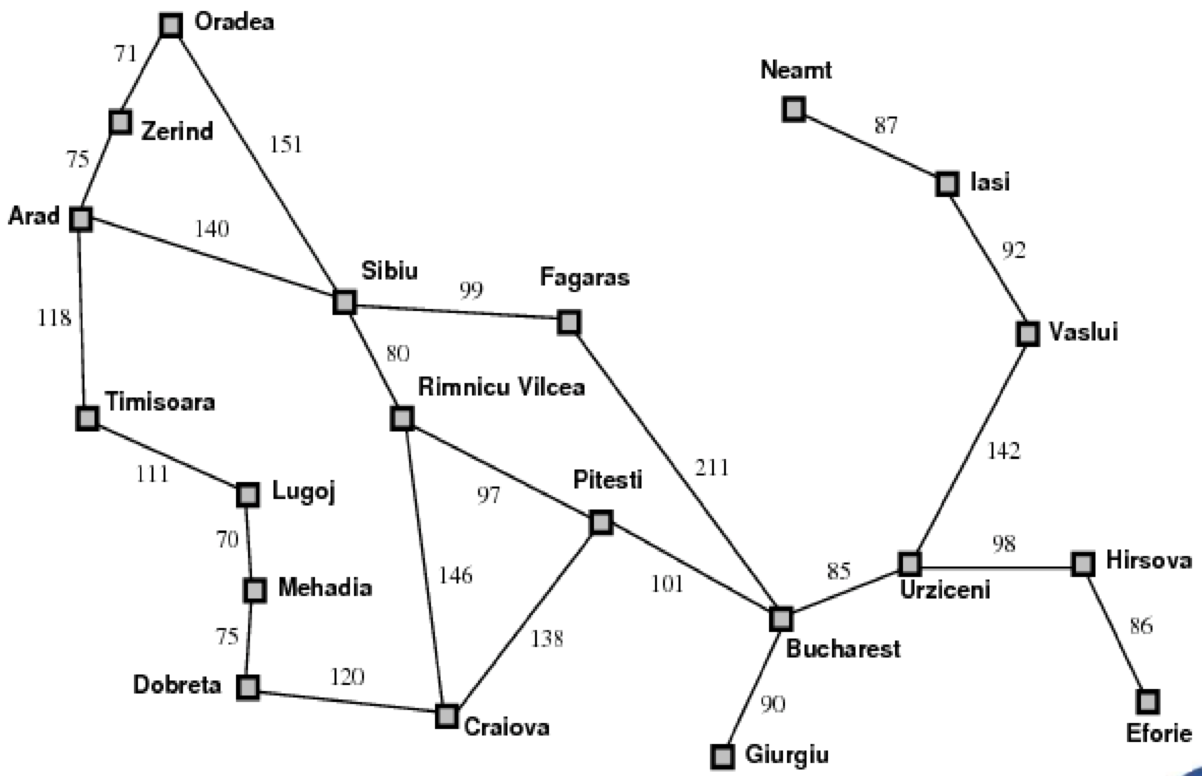
```
solve( Start, Solution) :-  breadthfirst( [ [Start] ], Solution).
breadthfirst( [ [Node | Path] | _ ], [Node | Path] ) :- goal(Node).
breadthfirst( [ [N | Path] | Paths], Solution) :-
                  bagof([M,N | Path],
                 ( s( N, M), \+ member( M, [N | Path] ) ),
                  NewPaths),       %NewPaths= acyclic       extensions of [N |
    Path]
conc( Paths, NewPaths, Paths1), !,
breadthfirst( Paths1, Solution);
breadthfirst(Paths, Solution). %Case that N has no successor

conc( [], L, L).
conc( [X| L1], L2, [X| L3]) :- conc( L1, L2, L3).
```

Summary

- Problem solving strategies
  – Problem formulation
  – State –space
  – Search problem
- DFS
  – Algorithm
  – Algorithm for closed set
  – Example
- BFS
  – Algorithm
  – Example

Check your understanding

- Implement the route planner using DFS to travel from Arad to Brucharest



Check your understanding

- Find the solution for the given initial and final states using BFS