G52APT
AI Programming Techniques

Lecture 8: Depth first search in Prolog

Brian Logan

School of Computer Science

bsl@cs.nott.ac.uk

Outline of this lecture

- recap: definition of a search problem
  - state spaces and search trees
- depth first search
- depth first search in Prolog
- loop detection

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

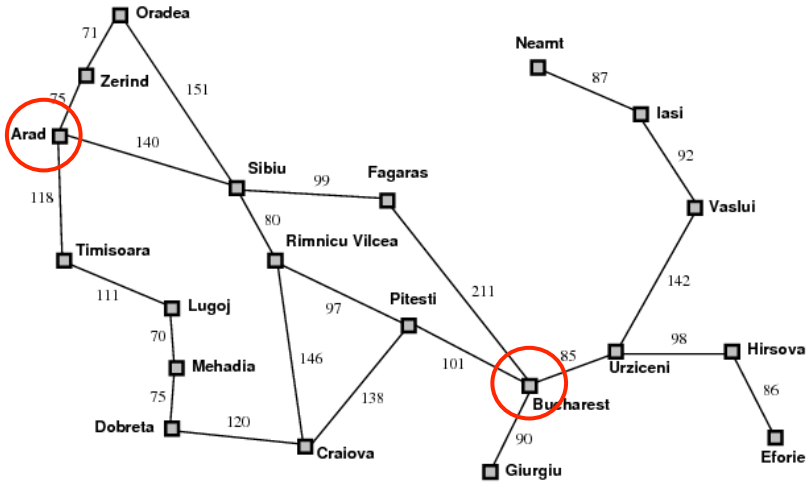
- depending on the number of solutions a problem has, there may be a single goal state or many goal states:
  - in the eight-puzzle there is a single correct configuration of tiles and a single goal state
  - in chess, there are many winning positions, and hence many goal states
- to avoid having to 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



Choice of operators and states

- if we consider other ways of solving the problem (e.g., flying or taking the train) we would need to change the set of operators as there are more actions we can perform in each state

- changing the operators often involves changing the set of states, e.g.:

  – if we allow train journeys, we would need to know *when* we were in a given city – 'being in city *X* at time *T*' – so that we could work out which trains we could take

  – if we allow taking a plane, we need to know which towns have airports

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 a 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

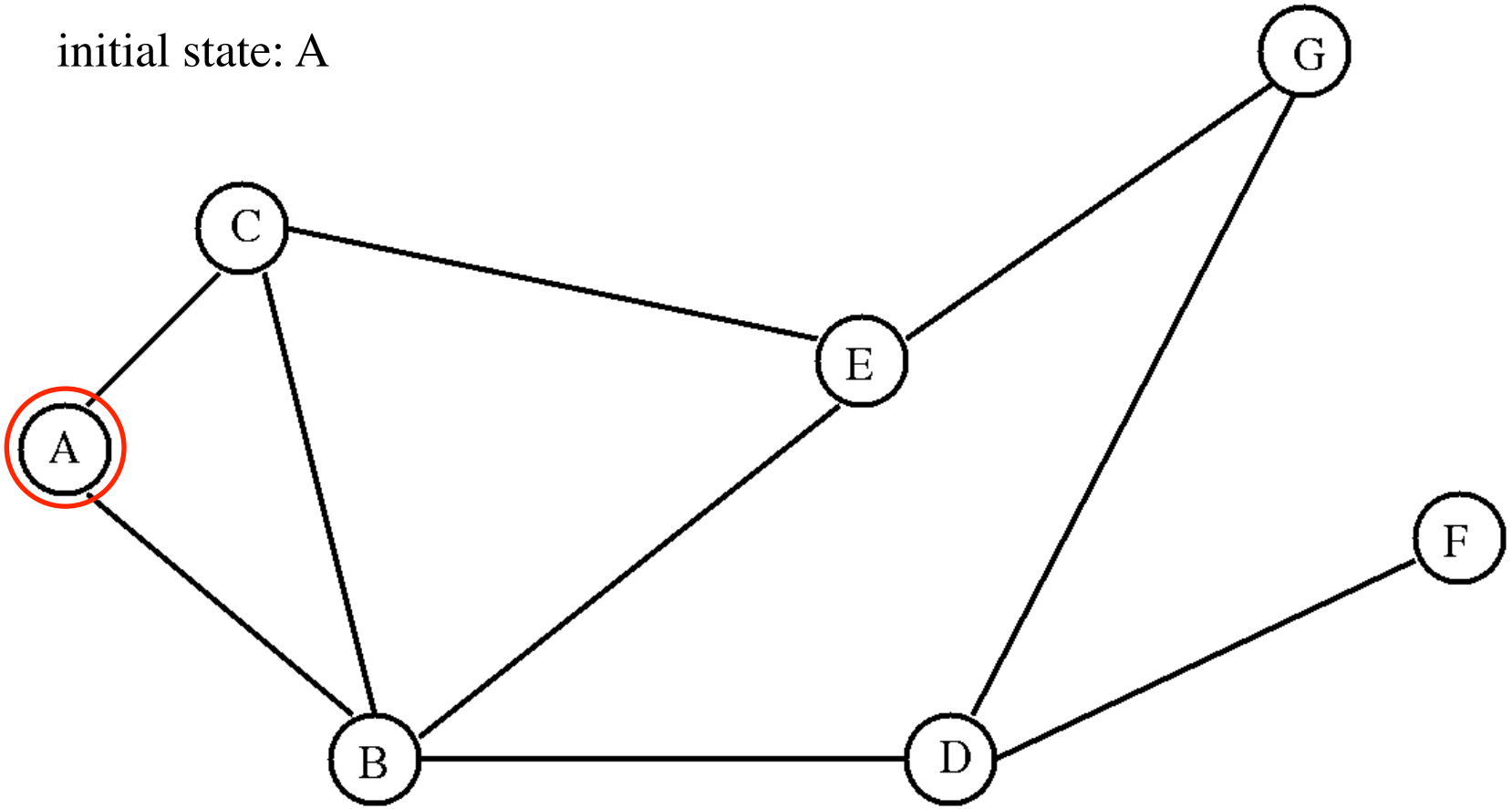
Exploring the state space

- *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

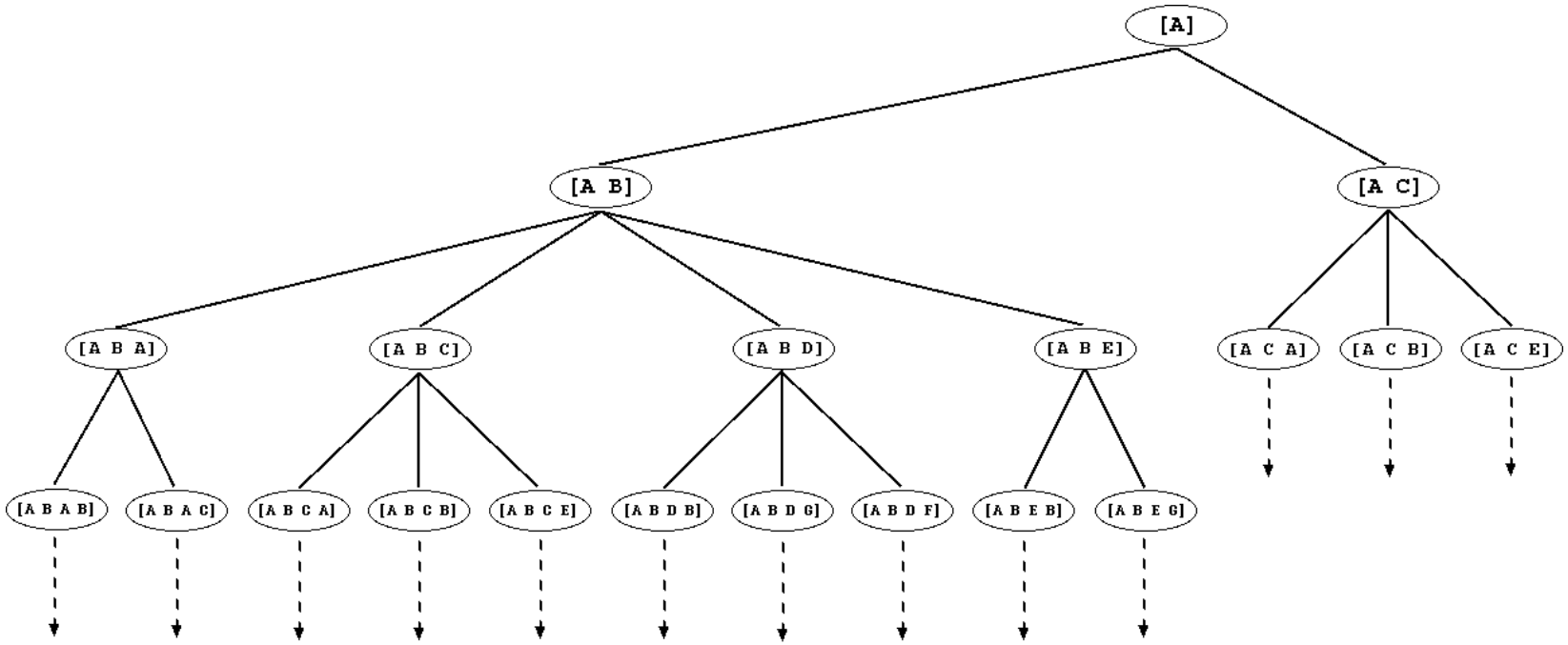
Search trees

- 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



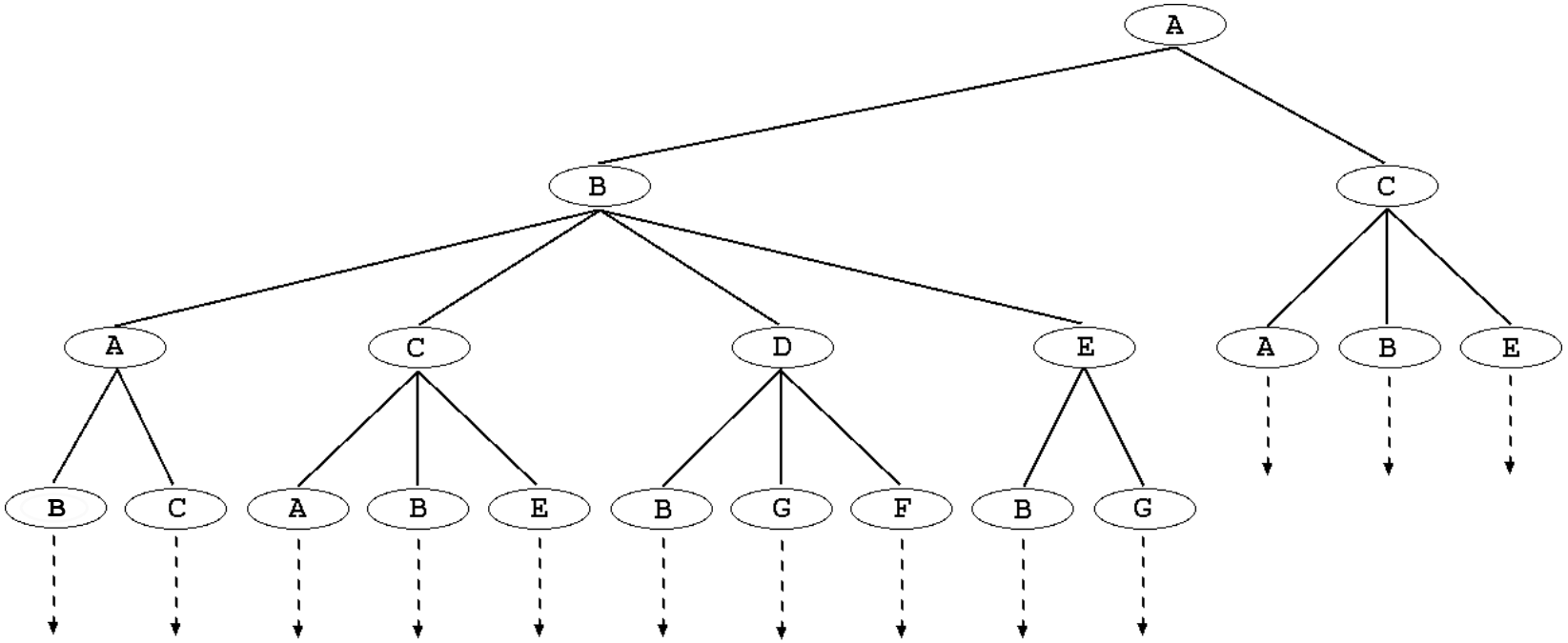
Example: search tree



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

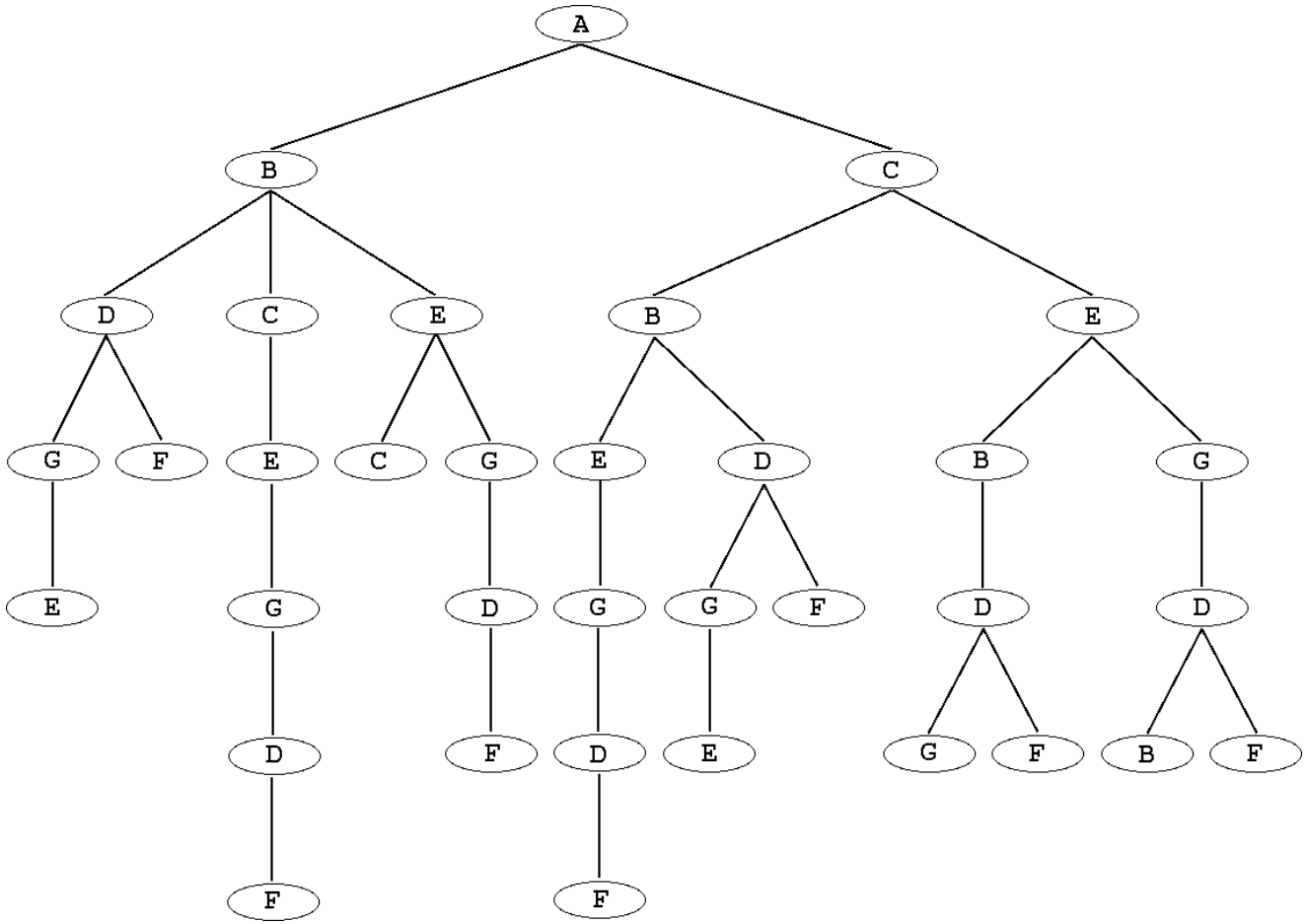
Example: labelling nodes



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*

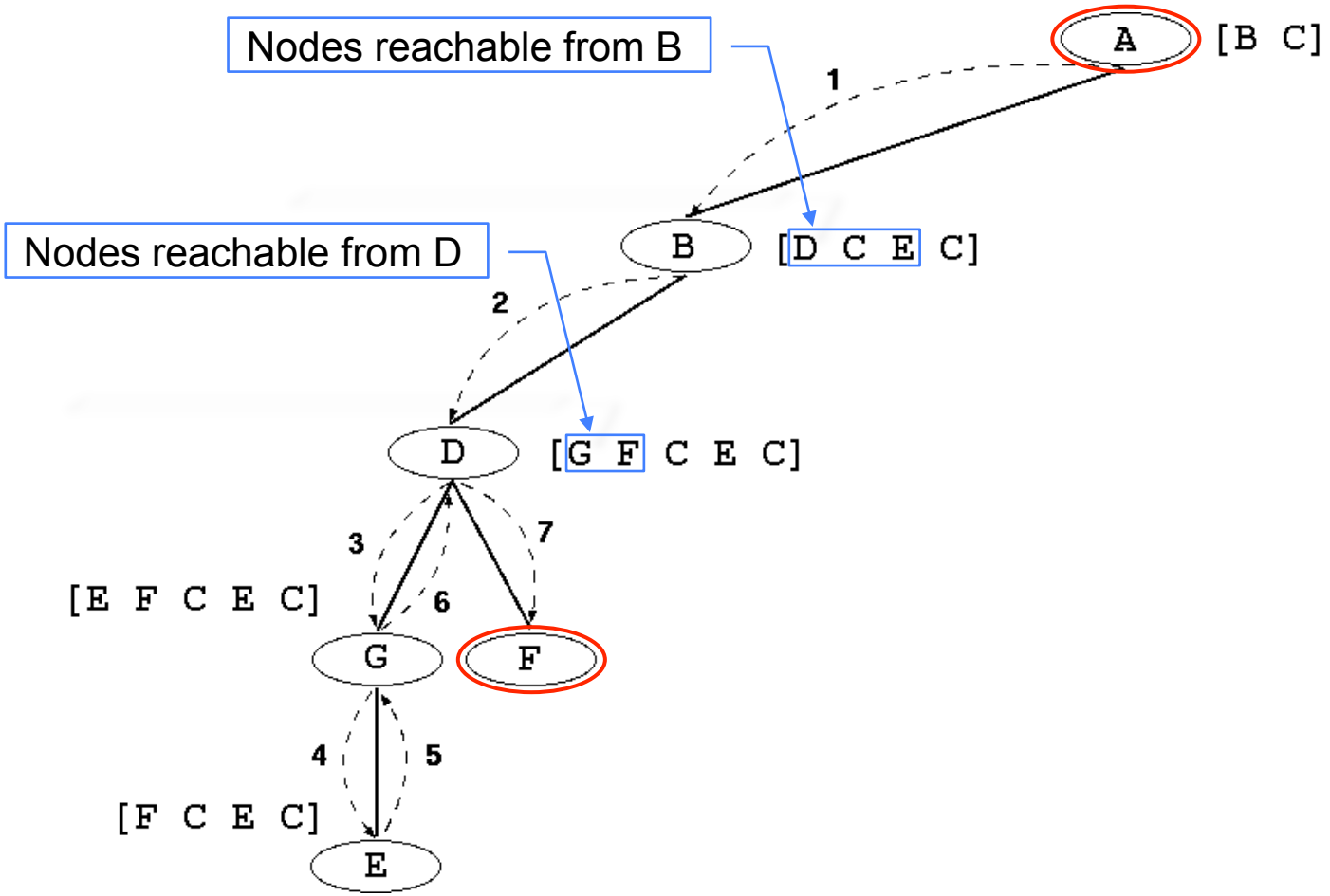
Example: eliminating loops



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



Properties of depth-first (tree) search

- space complexity is O(bm) where b is the branching factor and m is the maximum depth of the tree
- time complexity is O(b^m)
- not complete (unless the state space is finite and contains no loops)—we may get stuck going down an infinite branch that doesn't lead to a solution
- even if the state space is finite and contains no loops, the first solution found by depth-first search may not be the shortest

(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

Implementation choices

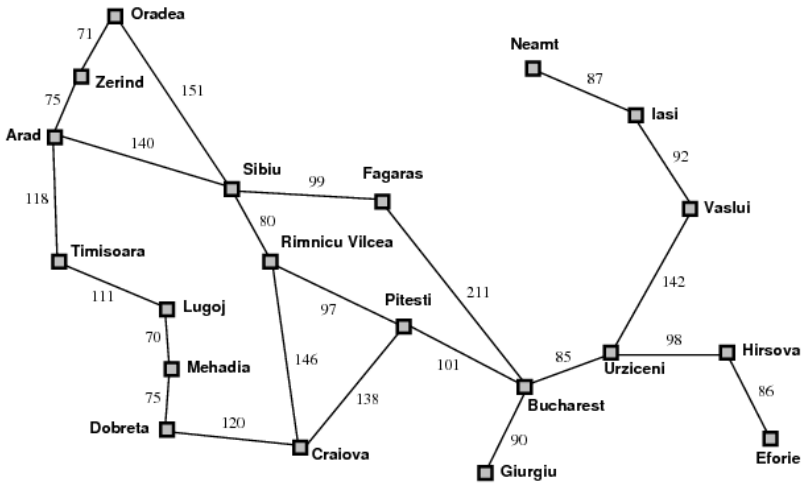
- choices depend (in part) on how generic we want our implementation to be
- two main alternatives:
  - **problem specific**: search algorithm is tailored to a particular problem or class of problems, e.g., route planning example
  - **generic**: complete problem description is given to a generic search algorithm
- focus initially on problem specific implementations – later we'll look at more generic approaches

Route planning example

```
route(X,Y,[drive(X,Y)]) :-
    road(X,Y).
route(X,Y,[drive(X,Z)|R]) :-
    road(X,Z),
    route(Z,Y,R).

road(arad,timisoara).
road(arad,sibiu).
road(arad,zerind).
road(zerind,oradea).
road(sibiu,fagaras).
road(sibiu,rimnicu_vilcea).

% etc ...
```



Example: route planning 1

- **states** are represented by atoms, but the representation of states is mostly implicit in the arguments to *route*/3
- **operators** are represented by complex terms (*drive*/2 terms) that represent an action of driving from one city to another, e.g.,

```
drive(arad,sibiu)
```

- **applicability of operators** is determined by the (implicit) current state and the existence of an appropriate road from the current state
- **paths** (and solutions) are represented by lists of operator terms, e.g.,

```
[drive(arad,sibiu),drive(sibiu,fagaras),
 drive(fagaras,bucharest)]
```

Example: route planning 2

- **nodes** are implicit – there is a single current node (corresponding to the current path), and other nodes are represented by backtrack points in the Prolog interpreter

- the **open list** is similarly implicit – represented by backtrack points (since there is no open list, space complexity is $O(m)$)

- the **closed list** is semi-implicit – Prolog remembers branches of the search tree that resulted in failure and won't try them again, but there is no list of previously visited states, for example

- see lecture 4 for how the search tree is explored

Extending the route planner

- the simple route planner given in previous lectures is only guaranteed to find a solution for directed acyclic graphs

- assumes that all roads are one-way, and the search space contains no loops

- we can extend it by adding an additional predicate that allows us to drive down a road in both directions, e.g.,

```
travel(X,Y) :- road(X,Y).
travel(X,Y) :- road(Y,X).
```

- this changes the implicit meaning of the *road*/2 unit clauses from 'there is a road *from X to Y*' to 'there is a road *between X and Y*'

Extending the route planner

```
route(X,Y,[drive(X,Y)]) :-
    travel(X,Y).
route(X,Y,[drive(X,Z)|R]) :-
    travel(X,Z),
    route(Z,Y,R).

travel(X,Y) :- road(X,Y).
travel(X,Y) :- road(Y,X).

road(arad,timisoara).
road(arad,sibiu).

% etc ...
```

Extending the route planner

- however making the edges undirected results in an infinite search tree
- as a result, the planner to fails to find any routes
- to extend it to general graphs, we need to add loop detection (i.e., a closed list)
- we can then keep track of which states (cities) we have visited before, and backtrack if expanding the current node with the operator would result in a loop

Adding a closed list

```
route(X,Y,R) :-
  route(X,Y,[X],R).

route(X,Y,V,[drive(X,Y)]) :-
  travel(X,Y).
route(X,Y,V,[drive(X,Z)|R]) :-
  travel(X,Z),
  \+ member(Z,V),
  route(Z,Y,[Z|V],R),
  Z \= Y.

travel(X,Y) :- road(X,Y).
travel(X,Y) :- road(Y,X).
```

Adding a closed list

```
route(X,Y,R) :-
  route(X,Y,[X],R).

route(X,Y,V,[drive(X,Y)]) :-
  travel(X,Y).
route(X,Y,V,[drive(X,Z)|R]) :-
  travel(X,Z),
  \+ member(Z,V),
  route(Z,Y,[Z|V],R),
  Z \= Y.

travel(X,Y) :- road(X,Y).
travel(X,Y) :- road(Y,X).
```

closed list

check that `Z` has not already been visited on this path

add `Z` to the closed list

only required if `Y` is not ground

Checking for visited states

- removing loops from paths makes the search tree finite, and allows the route planner to find routes in a graph with undirected edges
- however with this implementation, checking for loops requires $O(s)$ space and costs $O(s)$ at each node expansion, where s is the number of states
- if the state space (closed list) is large, it may be worth using, e.g, a tree to implement the closed 'list'

The next lecture

Depth limited/iterative deepening search in Prolog

Suggested reading:

- Bratko (2001) chapter 7