Artificial Intelligence

Lecture 2 – Problem Solving and Search

Outline

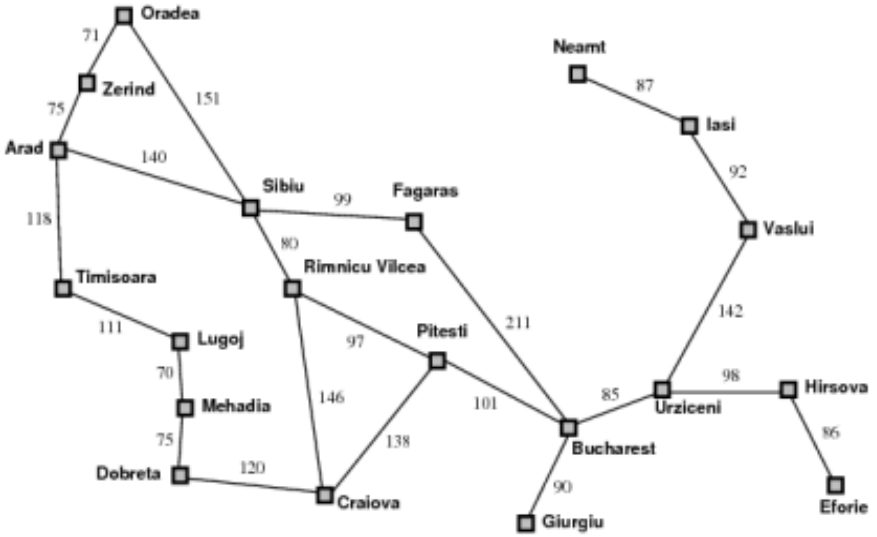
- Search problems
- Definition of a search problem
- State space
- Uninformed search methods
  - breadth first search
  - depth first search

Problem-solving and Search

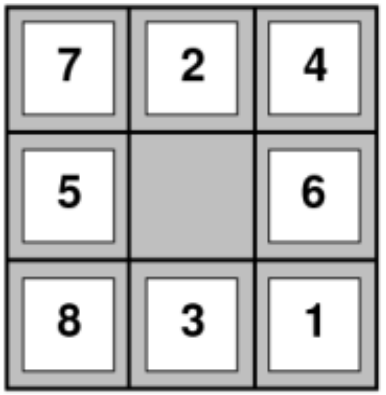
- Search is a universal problem-solving technique (weak method)
- Search (uninformed and informed) involves *systematic* trial and error exploration of alternative solutions
- Useful when the sequence of actions required to solve a problem is not known *a priori*
  - path finding problems, e.g. eight puzzle, travelling salesman problem
  - two player games, e.g. chess and checkers
  - constraint satisfaction problems, e.g. eight queens

Example Search Problems

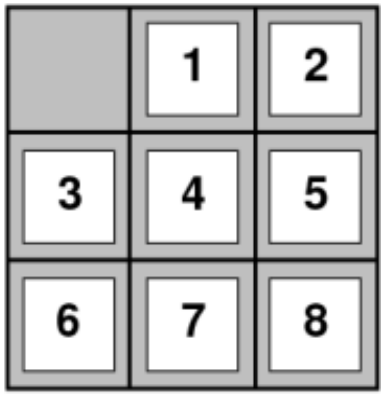
- Route planning



- Eight Puzzle



Start State



Goal State

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 solution 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 road
  - e.g, driving from Arad to Sibiu

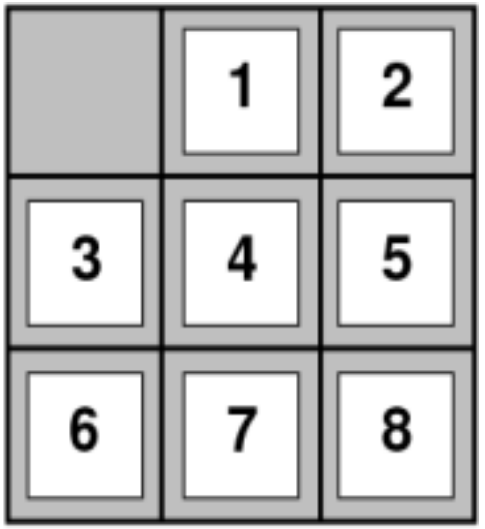
Choice of Operators and States

- 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
  - 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

Example: Eight Puzzle

- **states**: position of each tile and the blank tile, e.g., [(tile7 at 1, 3) (tile2 at 2, 3) … (tile1 at 3, 1)]
- **initial state**: initial positions of tiles
- **goal state**: tiles arranged in order, with the blank a top left
- **operators**: actions of moving a tile in a given position in each possible direction (up, down, left, right)

Start State



Goal State

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 and Solutions

- It is important to understand the difference between goals and solutions
- The *goal* is what we want to achieve
  - 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
  - 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 paths starting 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

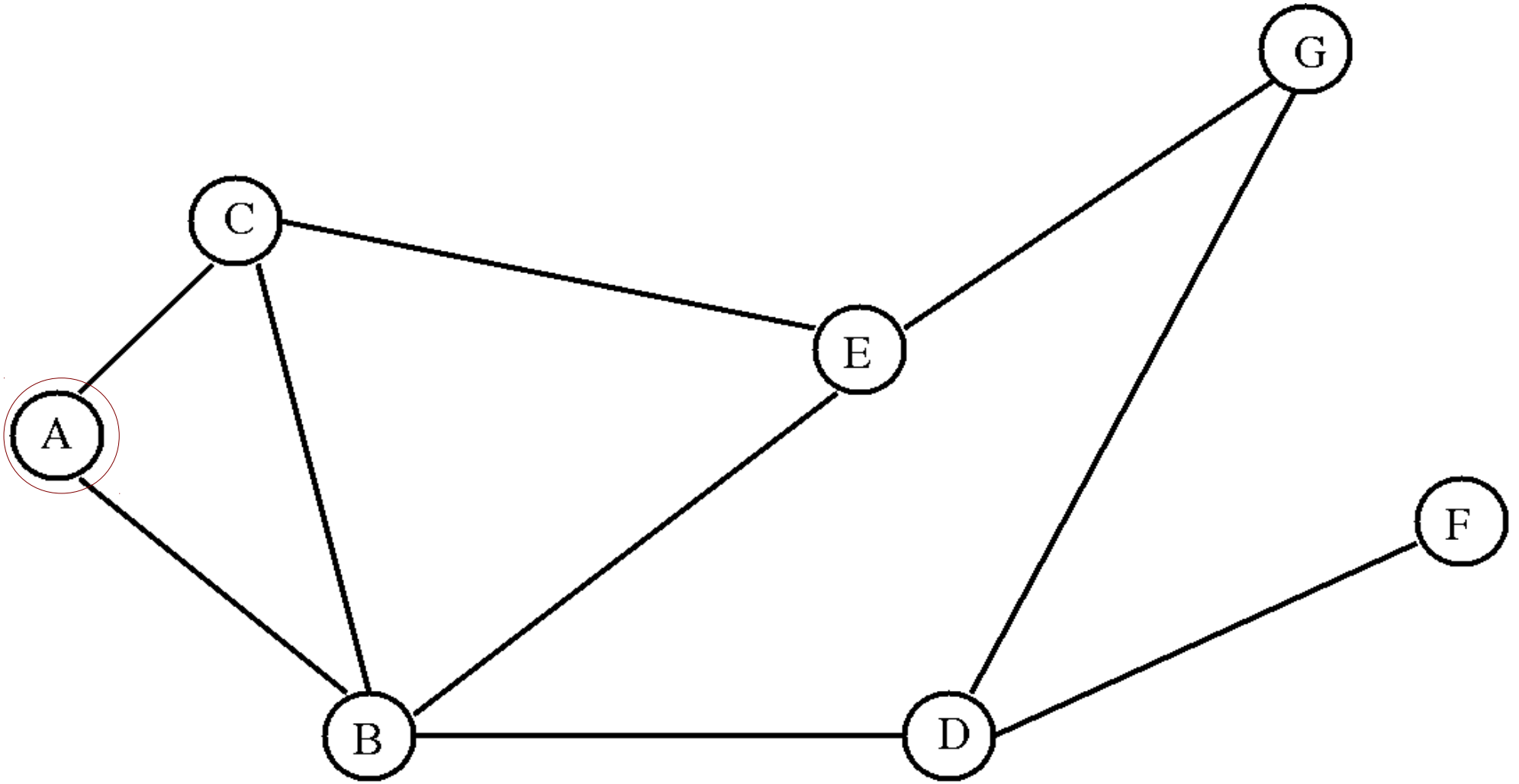
- *Search* is the process of exploring the state space to find a solution
- Exploration starts from the initial state
- The search procedure applies operators to the initial state to generate one or more new states which are believed to be nearer to a solution
- The search procedure is then applied recursively 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

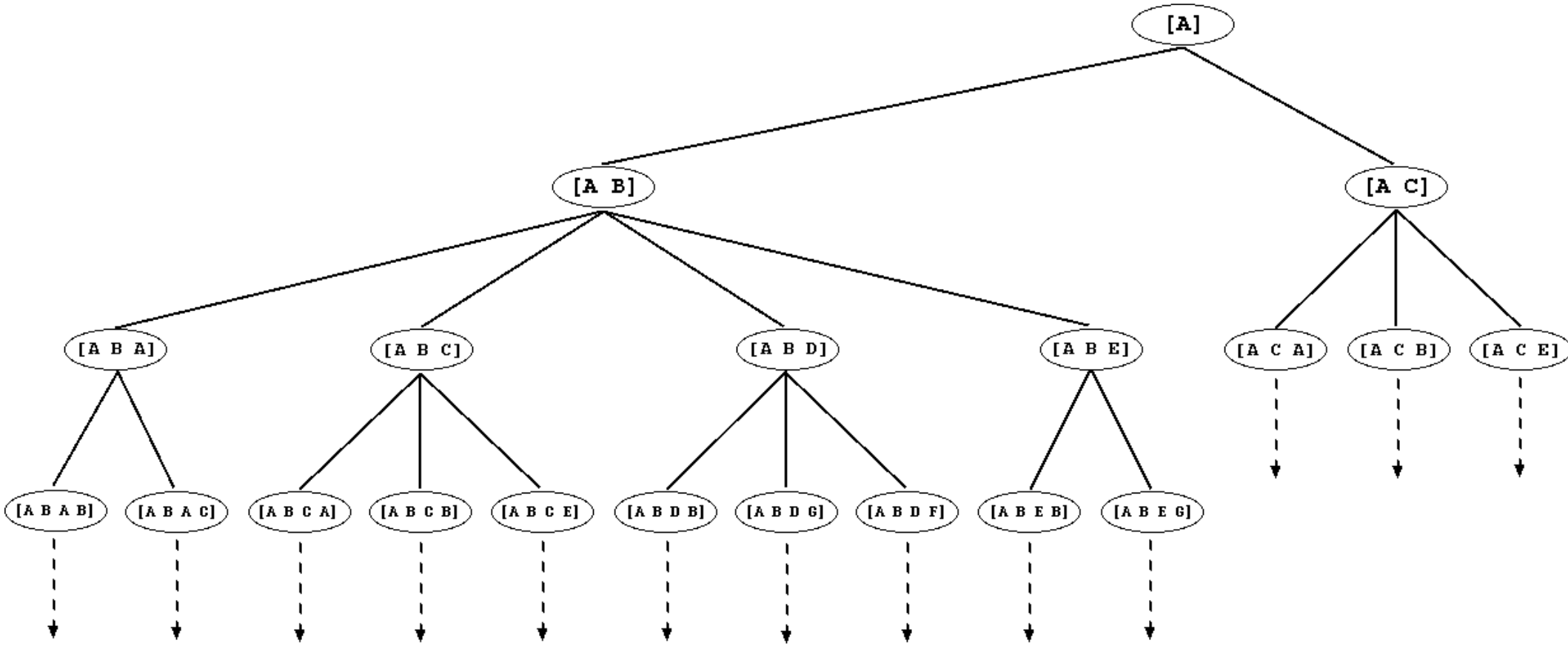
- The part of the state space that has been explored by a search procedure can be represented as a *search tree*
- Nodes in the search tree represent *paths* from the initial state (i.e., partial solutions) and edges represent operator applications
- The process of generating the children of a node by 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



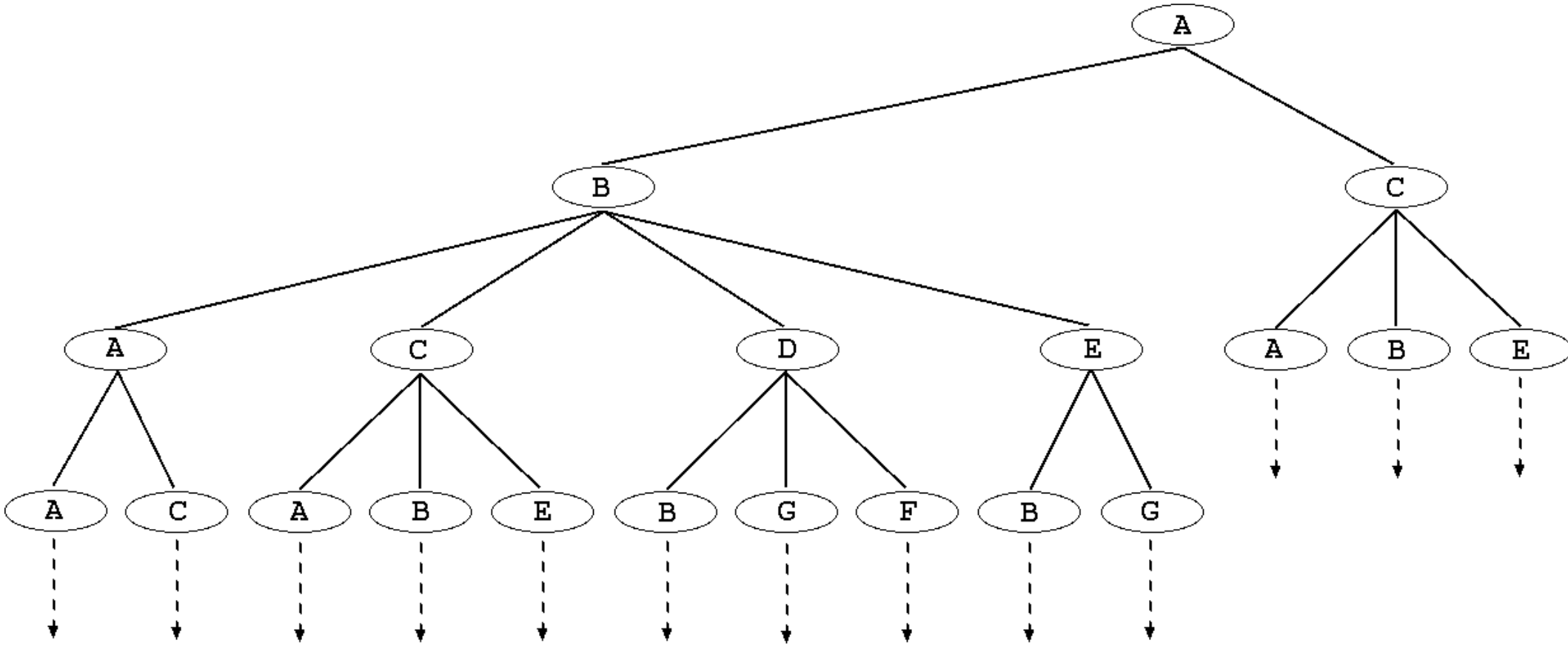
Example: Search Tree



States and 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

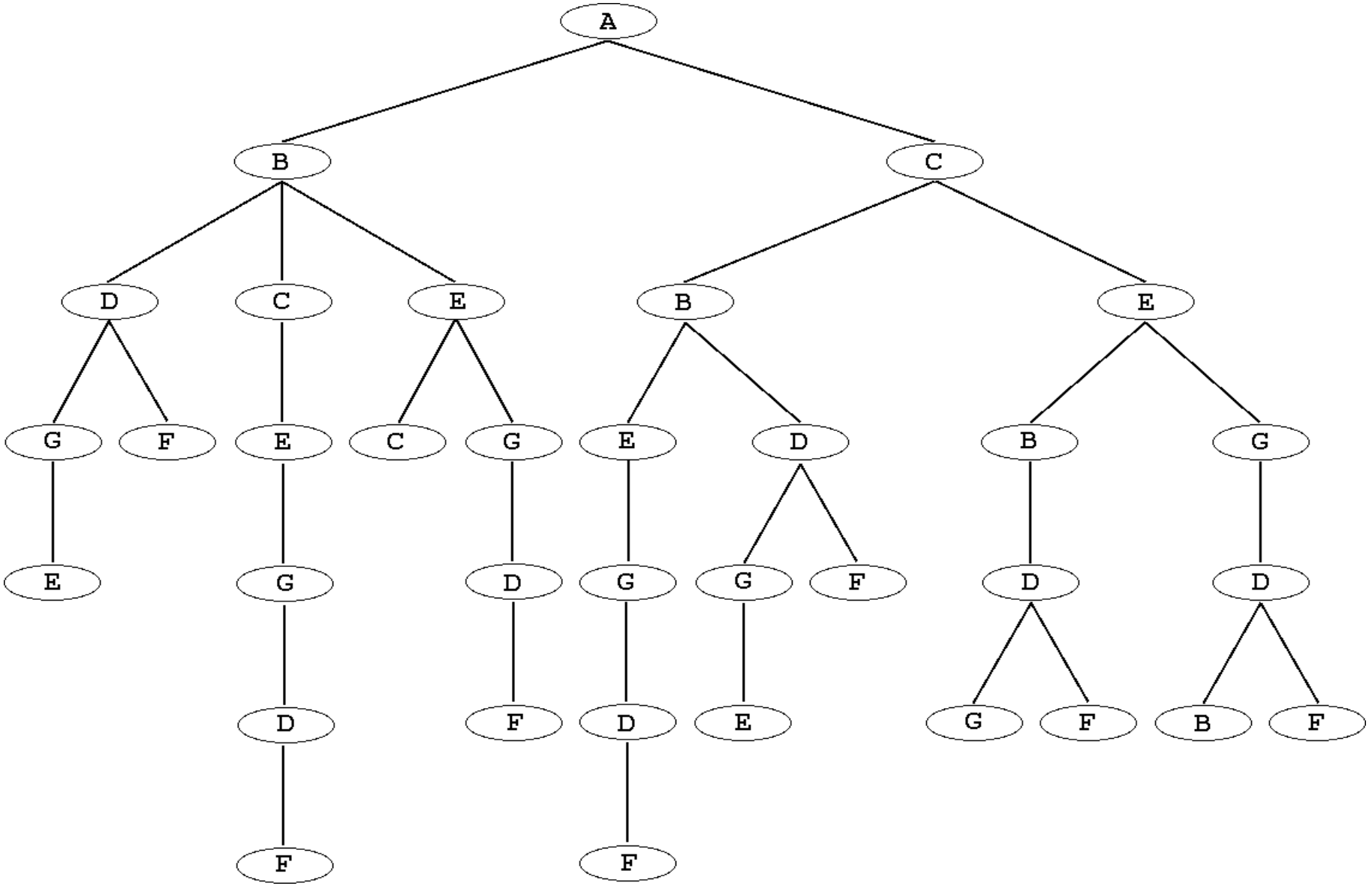
Example: Labelling Nodes



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 tree
- However eliminating loops can be *computationally expensive*

Example: Eliminating Loops



Solving Search Problems

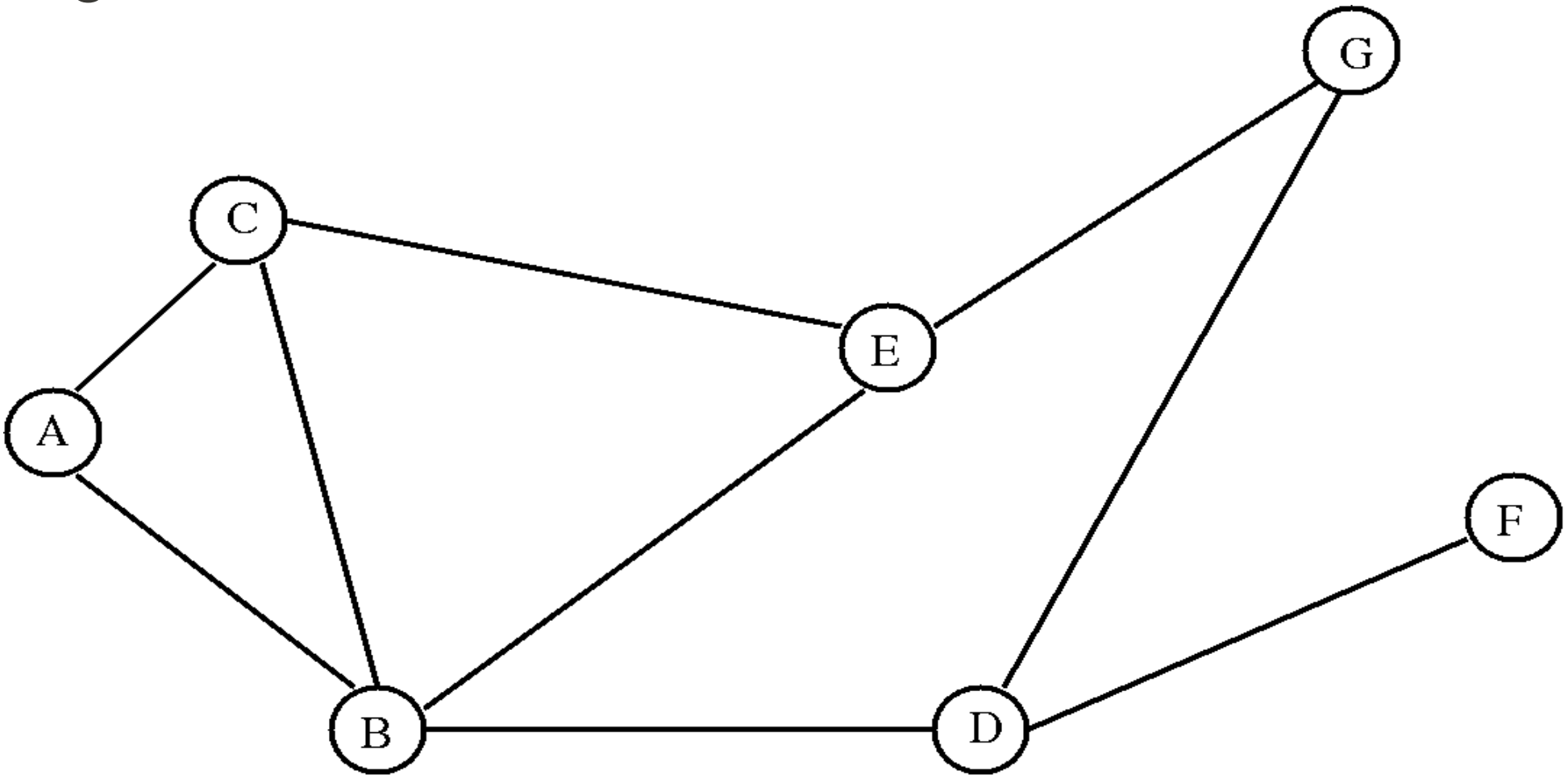
- Many different search techniques have been developed to explore a state space
  - uninformed (blind) search
  - informed (heuristic) search
  - GAs and other local search methods
  - stochastic search
- A search procedure which is guaranteed to find a solution (if one exists) is said to be *complete*

Breadth-first Search

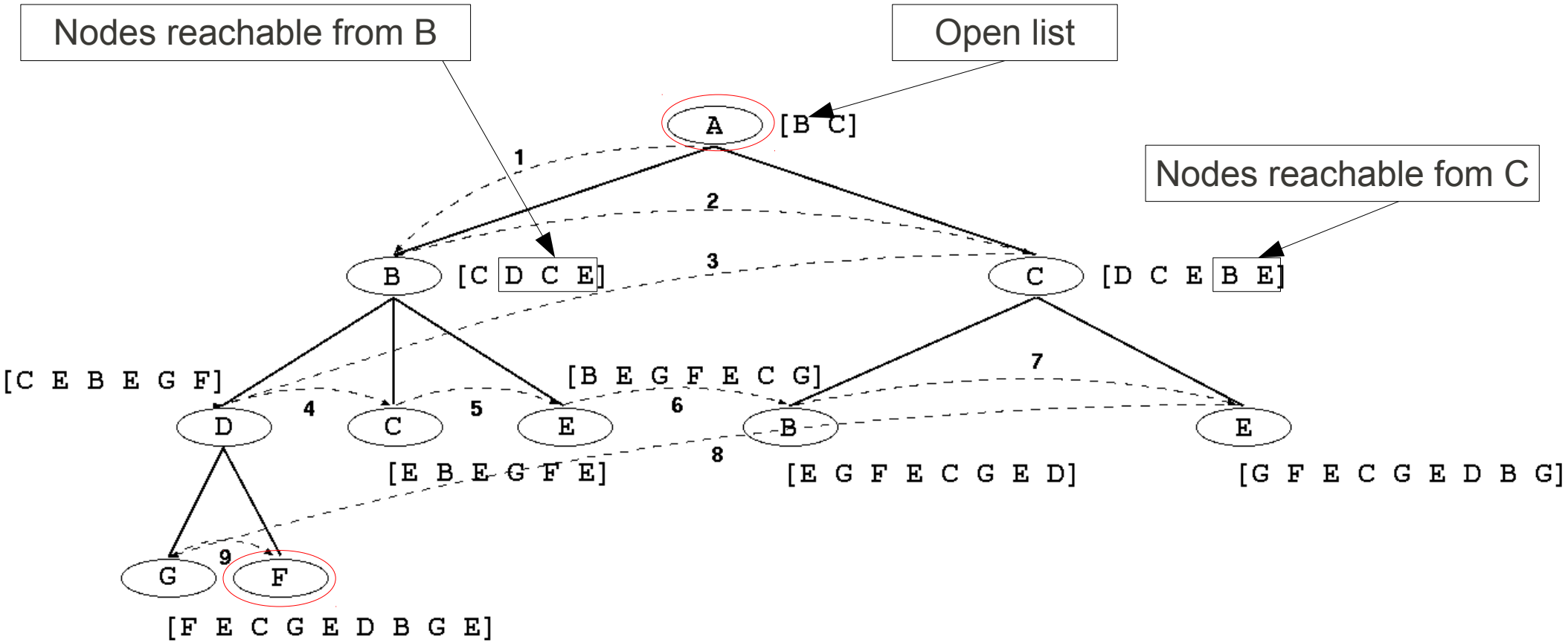
- Proceeds level by level down the search tree
- First explores all paths of length 1 from the root node, then all paths of length 2, length 3 etc.
- Starting from the root node (initial state) explores all children of the root node, left to right
- If no solution is found, expands the first (leftmost) child of the root node, then expands the second node at depth 1 and so on …

Example: Simple Route Planning

- initial state: A
- goal state: F

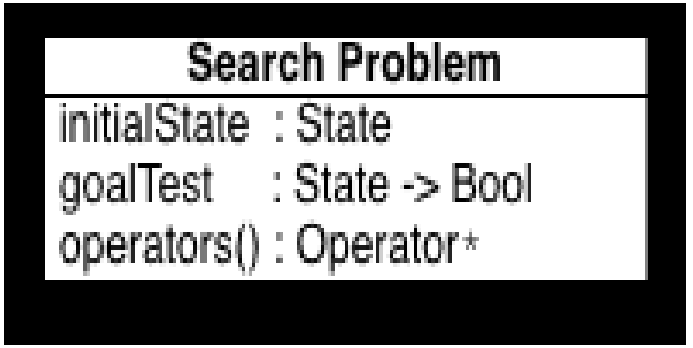


Example: Breadth-first Search

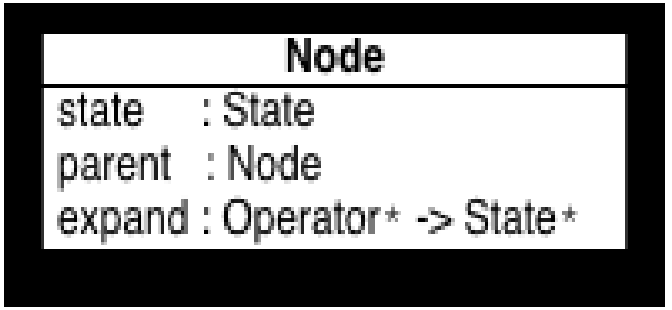


State, Search Problem & Node

```
// a search problem
class SearchProblem{
    public State initialState();
    public booleangoalTest(State s);
    public List<Op> operators();
}
```




```
// a search tree node
class Node{
    public State state();
    public Node parent();
    public List<Node> expand(List<Op> ops);
}
```



Breadth-first Search Algorithm

```
// pseudocodeimplementing breadth-first search
public Node breadthFirstSearch(SearchProblemproblem) {
    List<Node> nodes
       = new LinkedList<Node>(new Node(problem.initialState()))
    while(true) {
        if (nodes.size() == 0) then { return failure }
        Node node = nodes.removeFirst()
        if (problem.goalTest(node.state()) then { return node }
        // Note that new nodes are added to the end of the queue
        nodes.addAllToEnd(node.expand(problem.operators())
    }
}
```

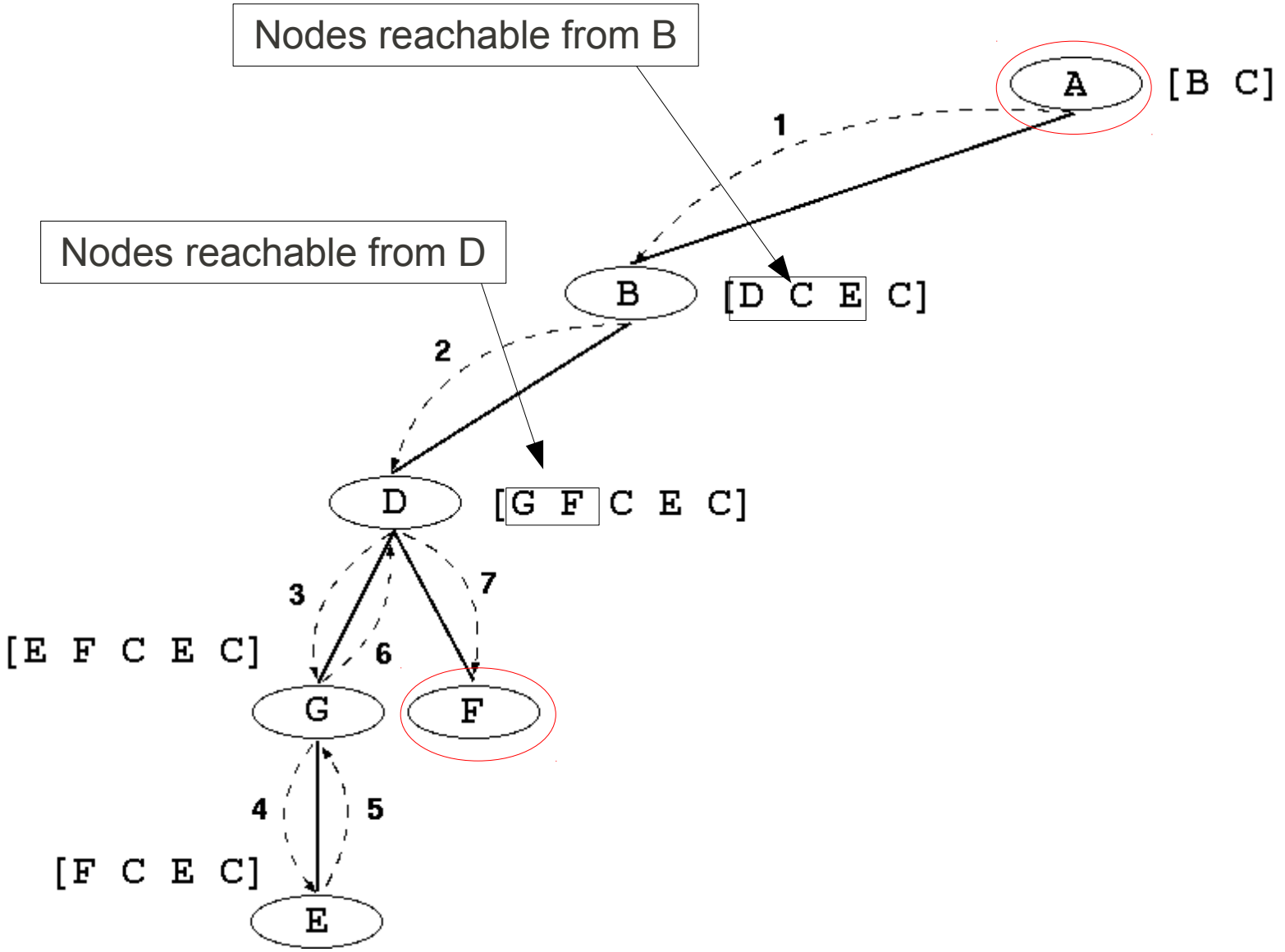
Properties of Breadth-first Search

- Breadth-first search is *complete* (even if the state space is infinite or contains loops)
- It is guaranteed to find the solution requiring the smallest number of operator applictions
- Time and space complexity is $O(b^d)$ where d is the depth of the shallowest solution
- Severely space bound in practice, and often runs out of memory very quickly

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 Algorithm

```
// pseudocodeimplementing depth-first search
public Node depthFirstSearch(SearchProblemproblem) {
   List<Node> nodes
      = new LinkedList<Node>(new Node(problem.initialState()))
   while(true) {
      if (nodes.size() == 0) then { return failure }
      Node node = nodes.removeFirst()
      if (problem.goalTest(node.state()) then { return node }
      // Note that new nodes are added to the front of the queue
      nodes.addAllToFront(node.expand(problem.operators())
   }
}
```

Properties of Depth-first Search

- Depth-first search requires much less memory than breadth-first search - space complexity is *O(bm)* where *m* is the maximum depth of the tree
- Time complexity is *O(bm)*
- However depth-first search is not complete (unless the state space is finite and no contains 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