Module 2

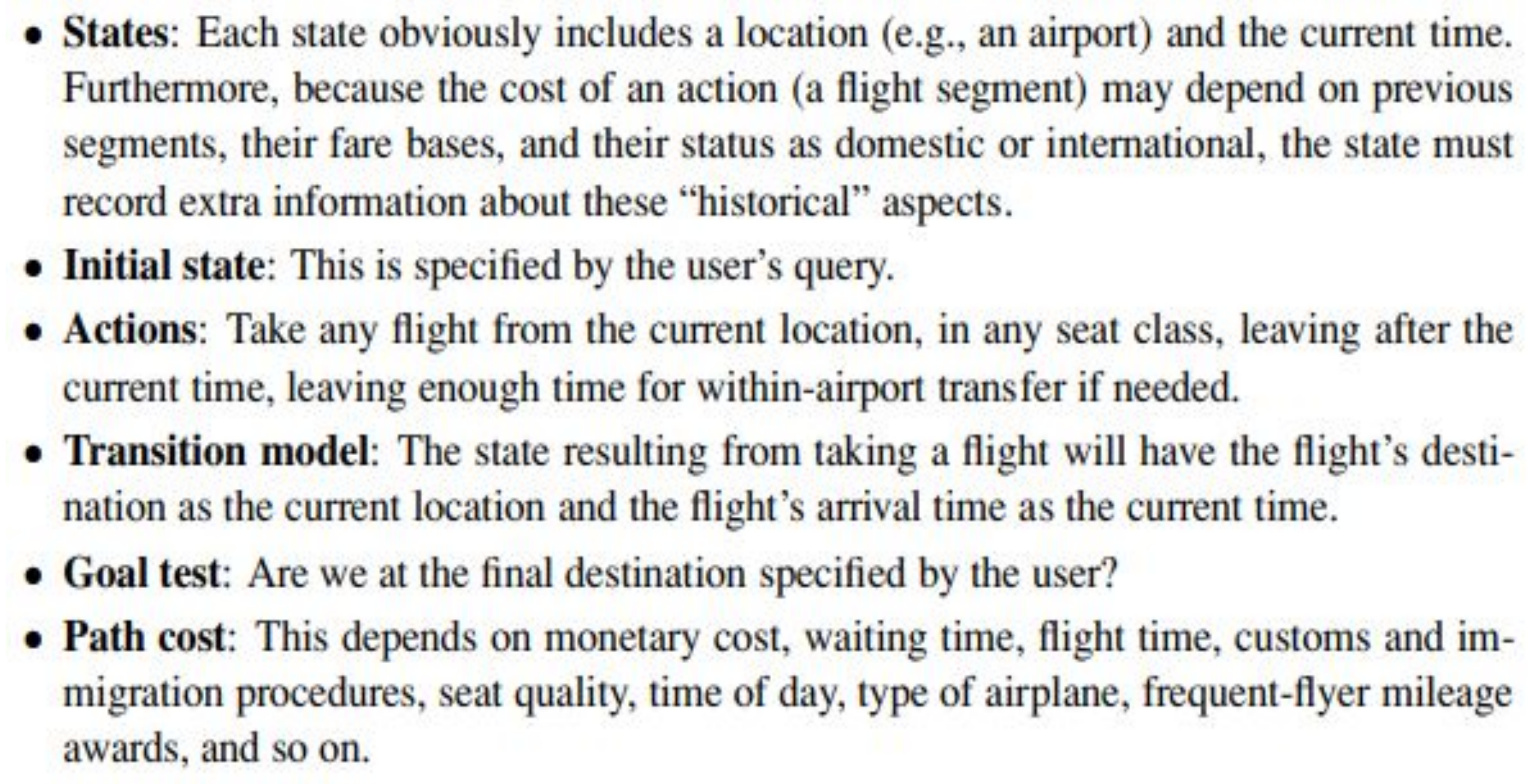
PART 2

Real-world problems

1. Route-finding problem

- Route-finding algorithms are used in a variety of applications.
- Some, such as Web sites and in-car systems that provide driving directions.
- Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications.

- Consider the airline travel problems that must be solved by a travel-planning Web site:

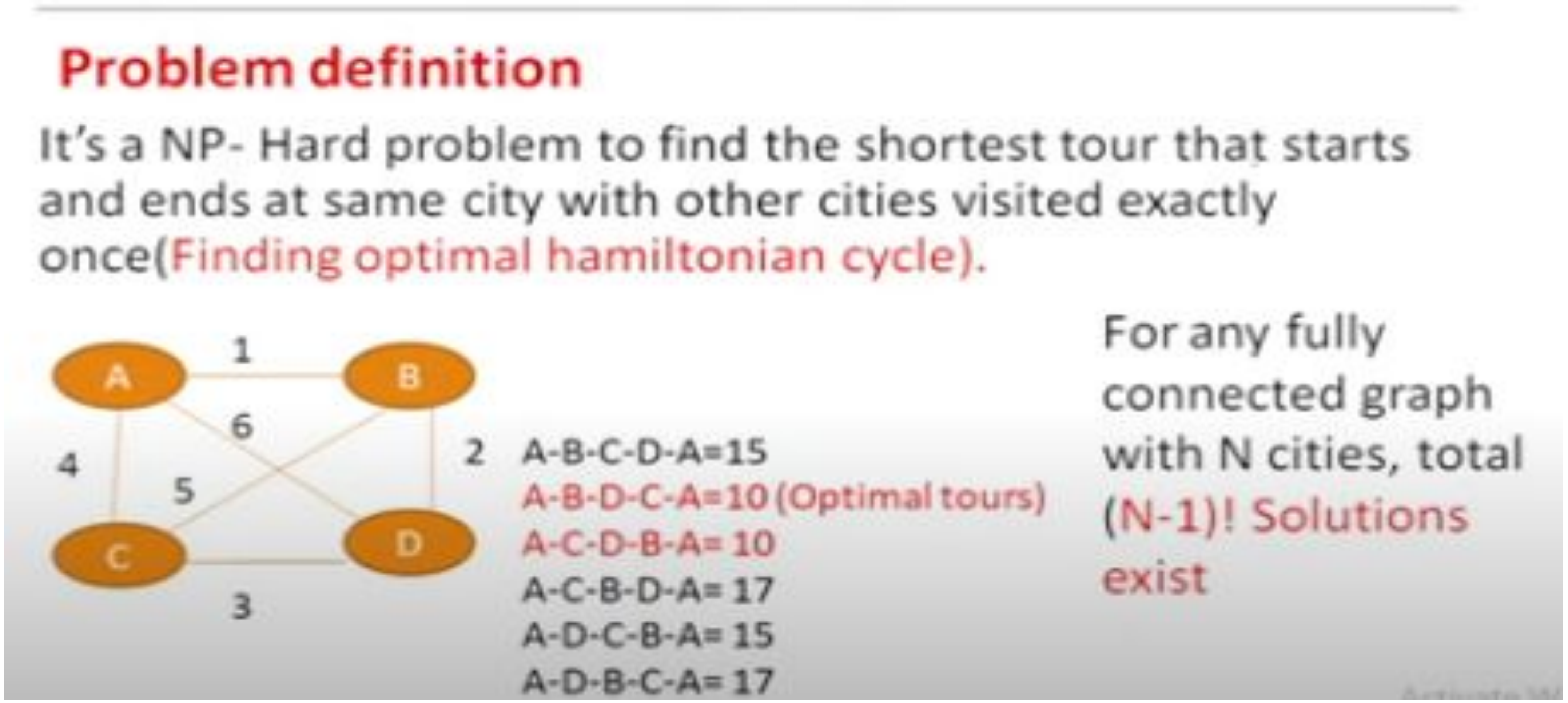


- **States**: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.
- **Initial state**: This is specified by the user's query.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- **Goal test**: Are we at the final destination specified by the user?
- **Path cost**: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

2. Touring problems

- Consider, for example, the problem “Visit every city in Romania at least once, starting and ending in Bucharest.”
- As with route finding, the actions correspond to trips between adjacent cities.
- The state space, however, is quite different.
- Each state must include not just the current location but also the set of cities the agent has visited.
- So the initial state would be In(Bucharest), Visited({Bucharest}), a typical intermediate state would be In(Vaslui), Visited({Bucharest, Urziceni, Vaslui}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

3. TSP

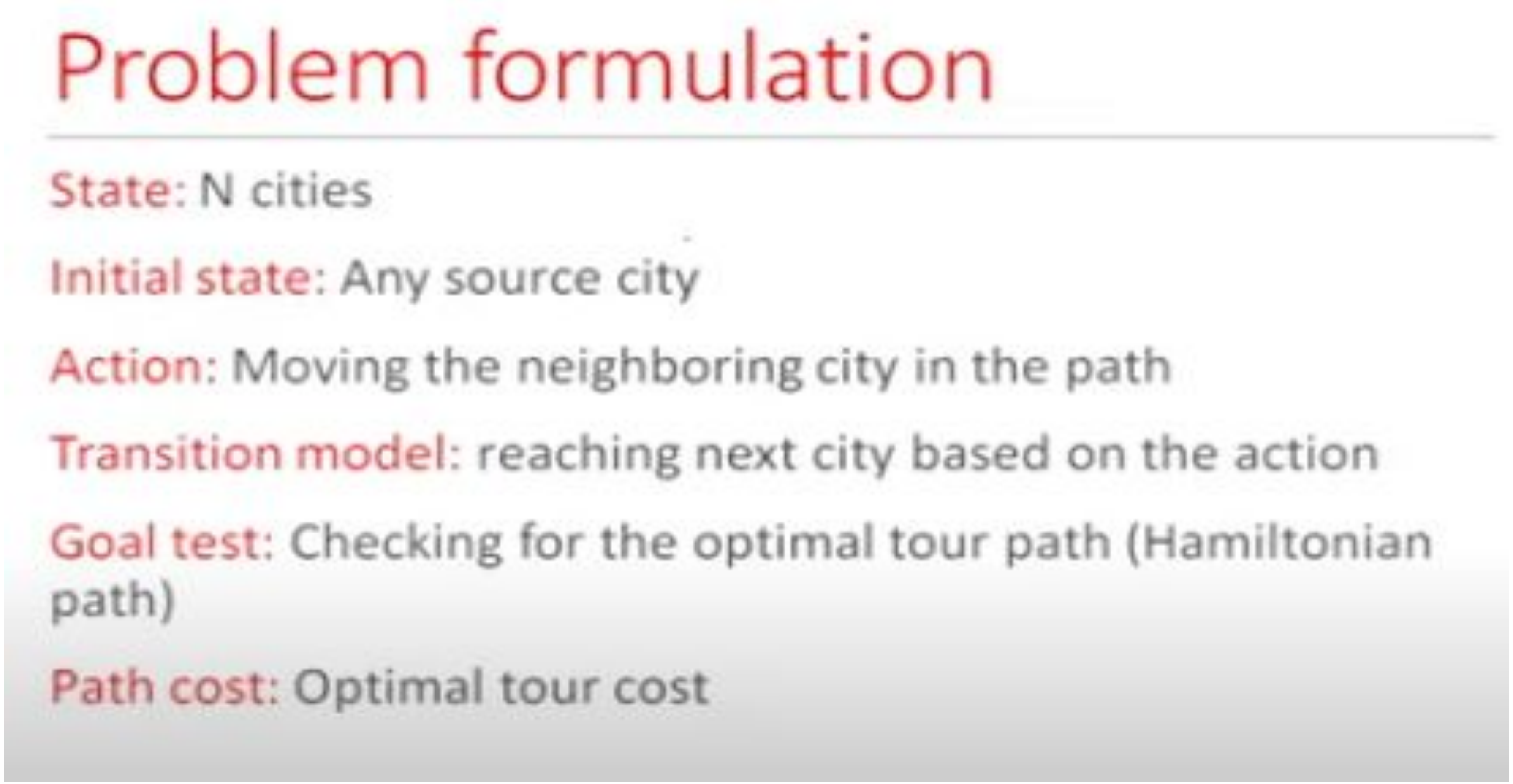


Problem definition

It's a NP- Hard problem to find the shortest tour that starts and ends at same city with other cities visited exactly once(Finding optimal hamiltonian cycle).

A-B-C-D-A=15
A-B-D-C-A=10 (Optimal tours)
A-C-D-B-A= 10
A-C-B-D-A= 17
A-D-C-B-A= 15
A-D-B-C-A= 17

For any fully connected graph with N cities, total (N-1)! Solutions exist



Problem formulation

State: N cities

Initial state: Any source city

Action: Moving the neighboring city in the path

Transition model: reaching next city based on the action

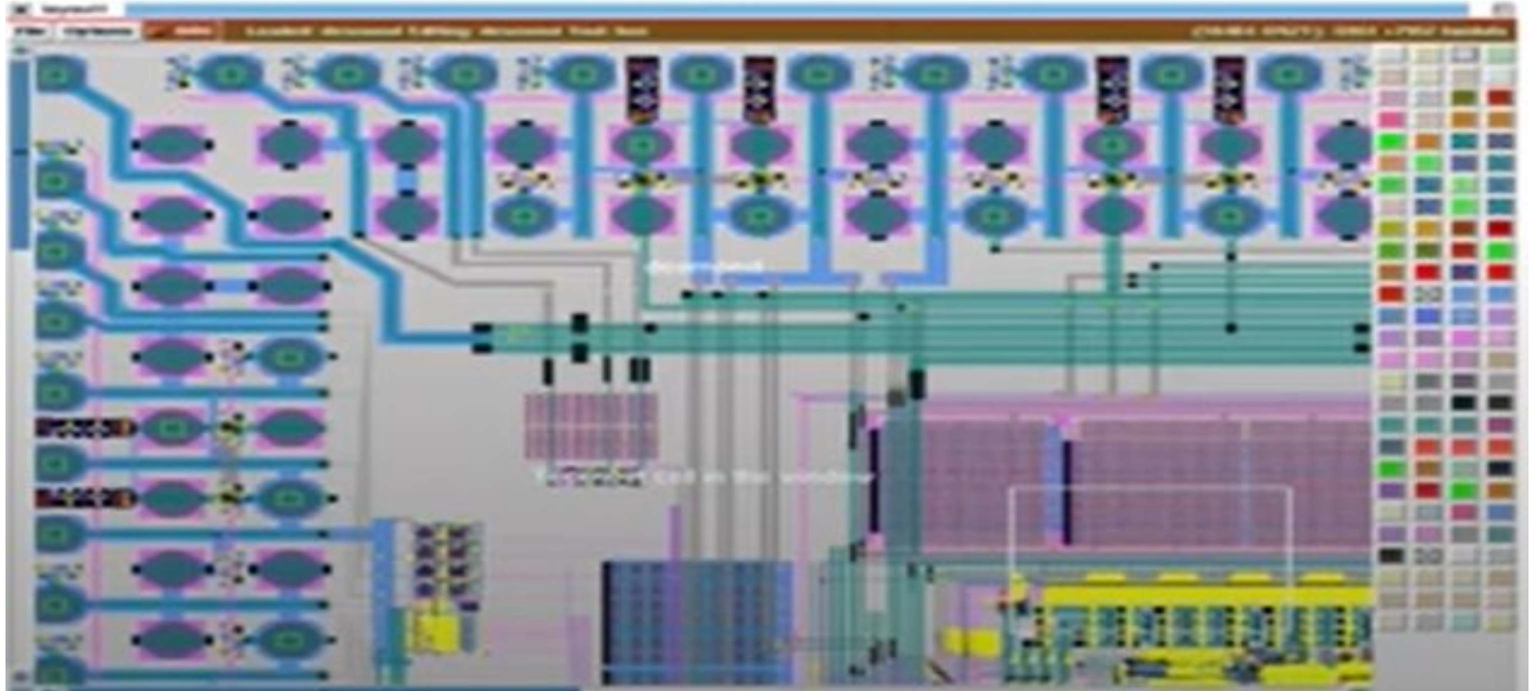
Goal test: Checking for the optimal tour path (Hamiltonian path)

Path cost: Optimal tour cost

- The traveling salesperson problem (TSP) is a touring problem in which each city must be visited exactly once.
- The aim is to find the shortest tour.
- The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms.
- In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

4. VLSI layout problem

- requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.
- The layout problem comes after the logical design phase and is usually split into two parts: cell layout and channel routing.



- In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells.

- The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells.
- Channel routing finds a specific route for each wire through the gaps between the cells.
- These search problems are extremely complex, but definitely worth solving.

5. Robot navigation

- is a generalization of the route-finding problem described earlier.
- Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.
- For a circular robot moving on a flat surface, the space is essentially two-dimensional.
- When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional.
- Advanced techniques are required just to make the search space finite.

6. Automatic assembly sequencing

- Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972).
- Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible.
- In assembly problems, the aim is to find an order in which to assemble the parts of some object.
- If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done.

- Another important assembly problem is protein design, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

Well-defined problems and solutions

- Together a **problem** is defined by
  - Initial state
  - Actions
  - Transition model
  - Goal test
  - Path cost
- The ***solution*** to a problem is
  An action sequence that leads from the initial state to a goal state.
- ***Optimal*** solution
  the solution with lowest path cost among all solutions.

SEARCHING FOR SOLUTIONS

- Having formulated some problems, we now need to solve them.

- A solution is an action sequence, so search algorithms work by considering various possible action sequences. This is called a search tree.

- The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem.

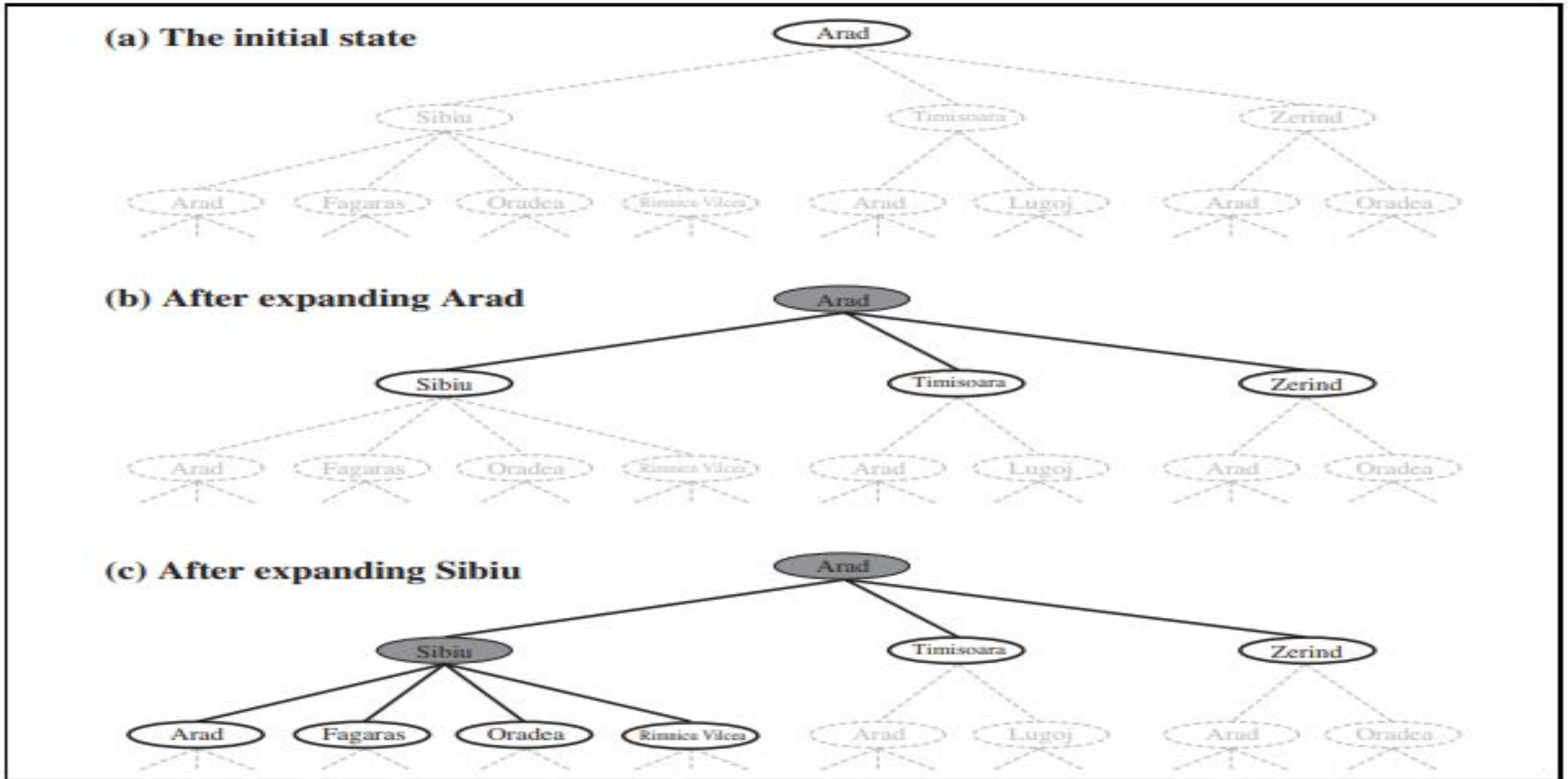


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

- Figure shows the first few steps in growing the search tree for finding a route from Arad to Bucharest.
- The root node of the tree corresponds to the initial state, In(Arad).
- The first step is to test whether this is a goal state.
- Then we need to consider taking various actions.
- We do this by **expanding** the current state; that is, applying each legal action to the current state, thereby **generating** a new set of states.
- In this case, we add three branches from the **parent node** In(Arad) leading to three new **child nodes**: In(Sibiu), In(Timisoara), and In(Zerind).
- Now we must choose which of these three possibilities to consider further.

- Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get In(Arad), In(Fagaras), In(Oradea), and In(RimnicuVilcea).
- We can then choose any of these four or go back and choose Timisoara or Zerind.
- Each of these six nodes is a **leaf node**, that is, a node with no children in the tree.
- The set of all leaf nodes available for expansion at any given point is called the **frontier**.
- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.

TREE SEARCH ALGORITHM

- Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**.

- We say that In(Arad) is a **repeated state** in the search tree, generated in this case by a **loopy path**.
- Considering such loopy paths means that the complete search tree for Romania is **infinite** because there is no limit to how often one can traverse a loop.
- Loops can cause certain algorithms to fail, making otherwise solvable problems unsolvable.
- Fortunately, there is no need to consider loopy paths.
- We can rely on more than intuition for this: because path costs are additive and step costs are nonnegative, a loopy path to any given state is never better than the same path with the loop removed.

- Loopy paths are a special case of the more general concept of **redundant paths**, which exist whenever there is more than one way to get from one state to another.
- Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long). Obviously, the second path is redundant—it’s just a worse way to get to the same state.
- If you are concerned about reaching the goal, there’s never any reason to keep more than one path to any given state, because any goal state that is reachable by extending one path is also reachable by extending the other.

- Algorithms that forget their history are doomed to repeat it.
- The way to avoid exploring redundant paths is to remember where one has been.
- To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set** (also known as the **closed list**), which remembers every expanded node.
- Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.

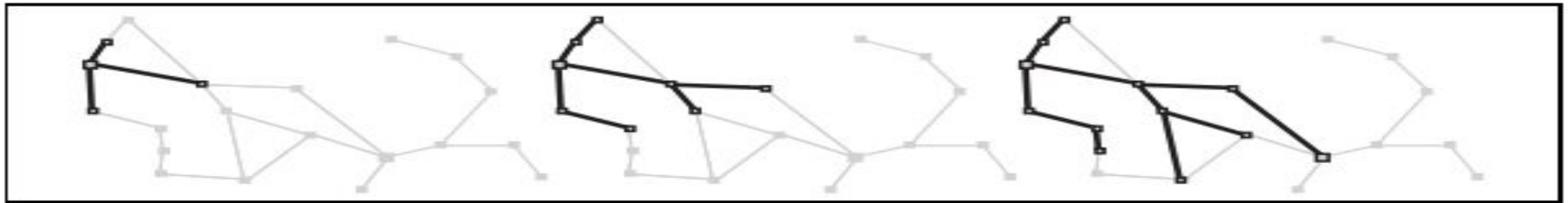
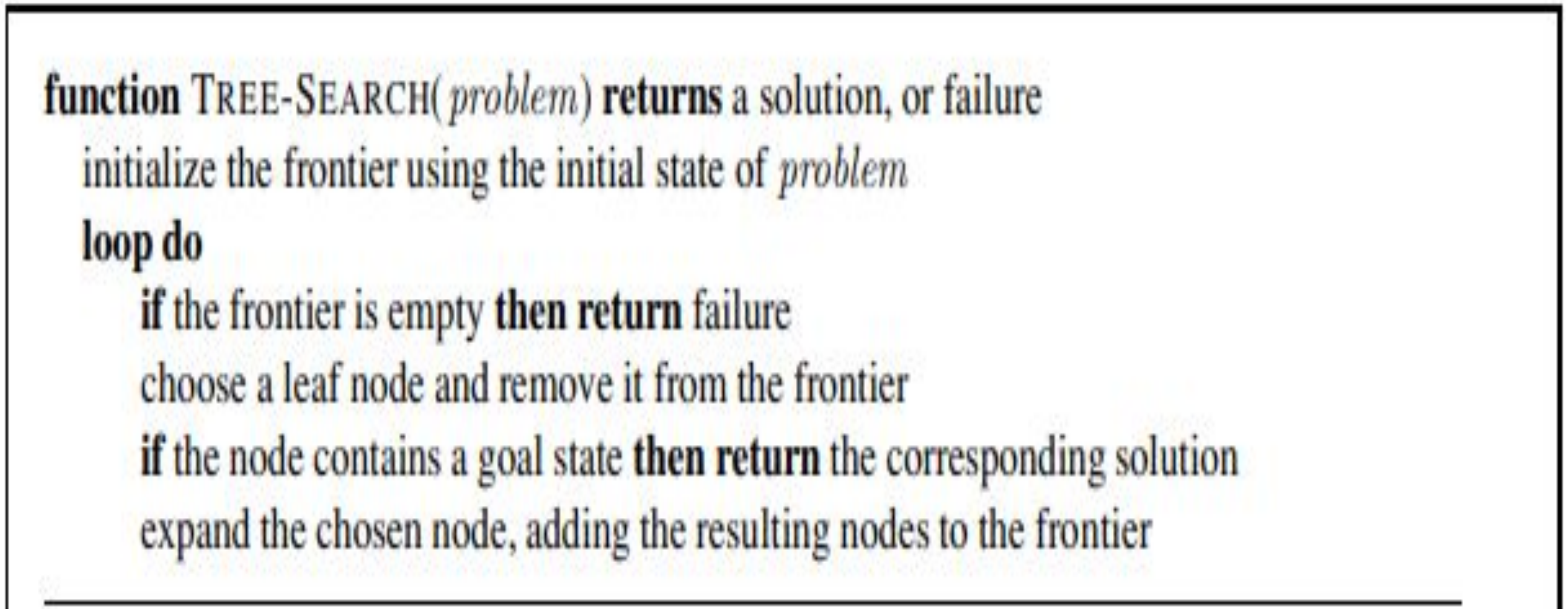


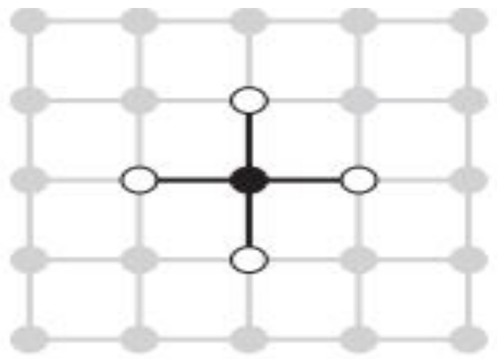
Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

The general TREE-SEARCH algorithm is shown informally in Figure .



```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

- In other cases, redundant paths are unavoidable.
- This includes all problems where the actions are reversible, such as route-finding problems and sliding-block puzzles.
- Route finding on a **rectangular grid** is a particularly important example in computer game.



- In such a grid, each state has four successors, so a search tree of depth d that includes repeated states has 4^d leaves; but there are only about $2d^2$ distinct states within d steps of any given state.
- For d = 20, this means about a trillion nodes but only about 800 distinct states. Thus, following redundant paths can cause a tractable problem to become intractable. This is true even for algorithms that know how to avoid infinite loops.

- The algorithm has another nice property: the frontier separates the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier.

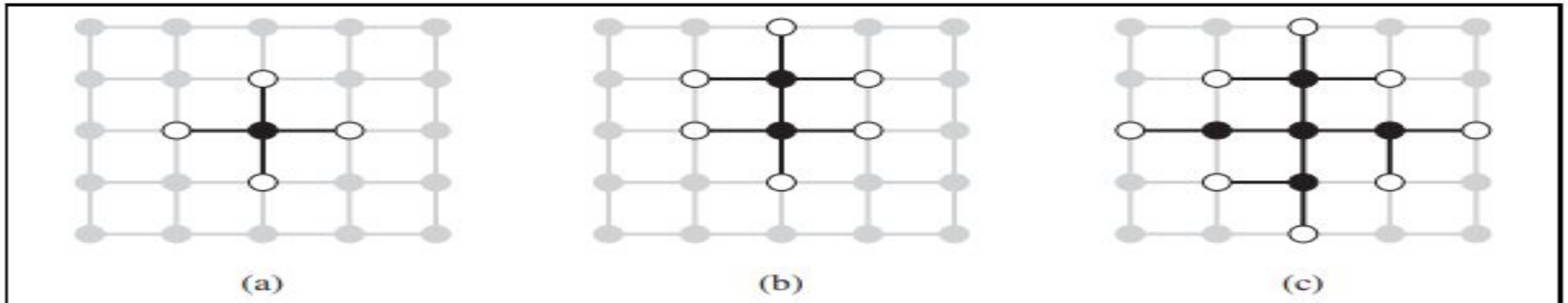
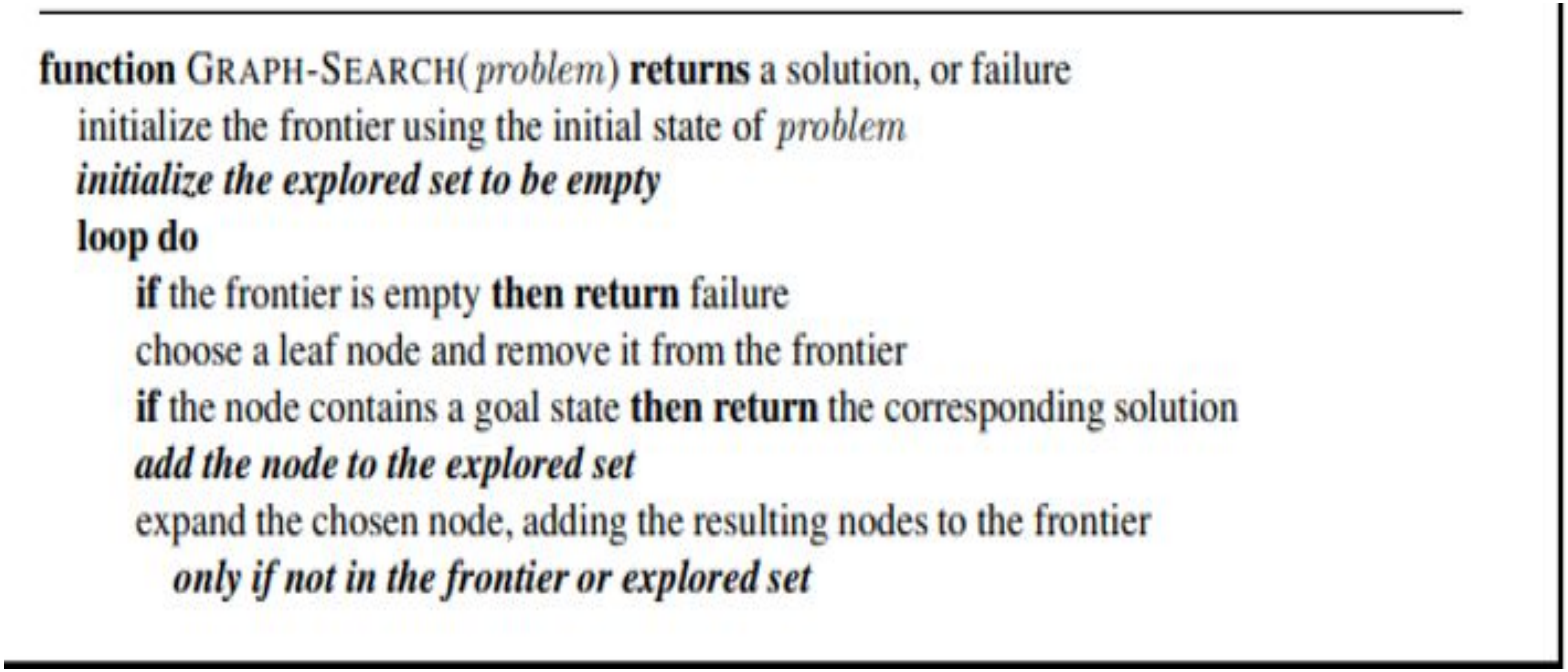


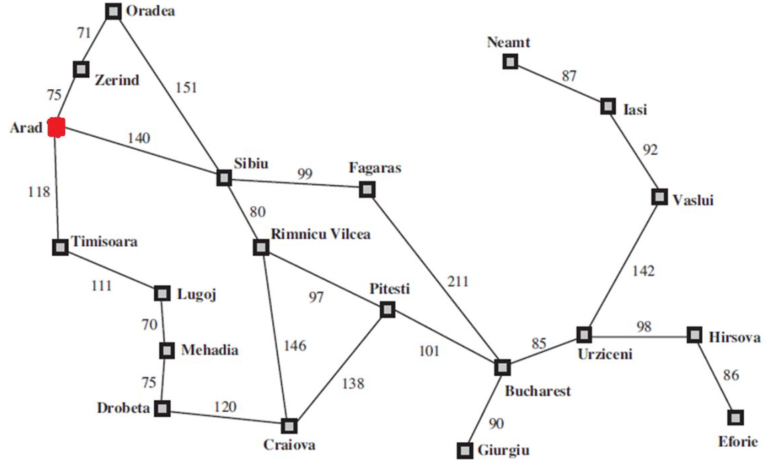
Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

- As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.

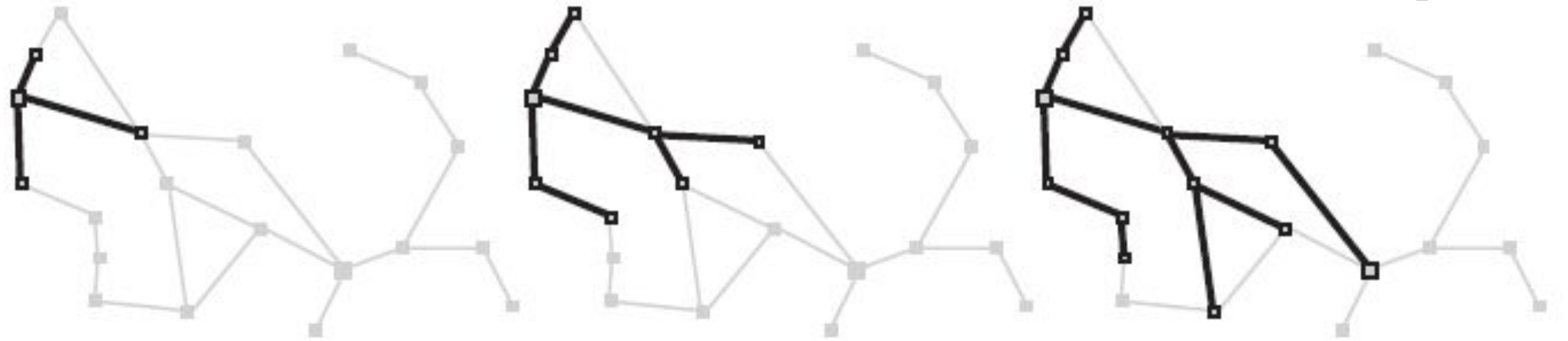
The new algorithm, called GRAPH-SEARCH, is shown informally in Figure .



```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

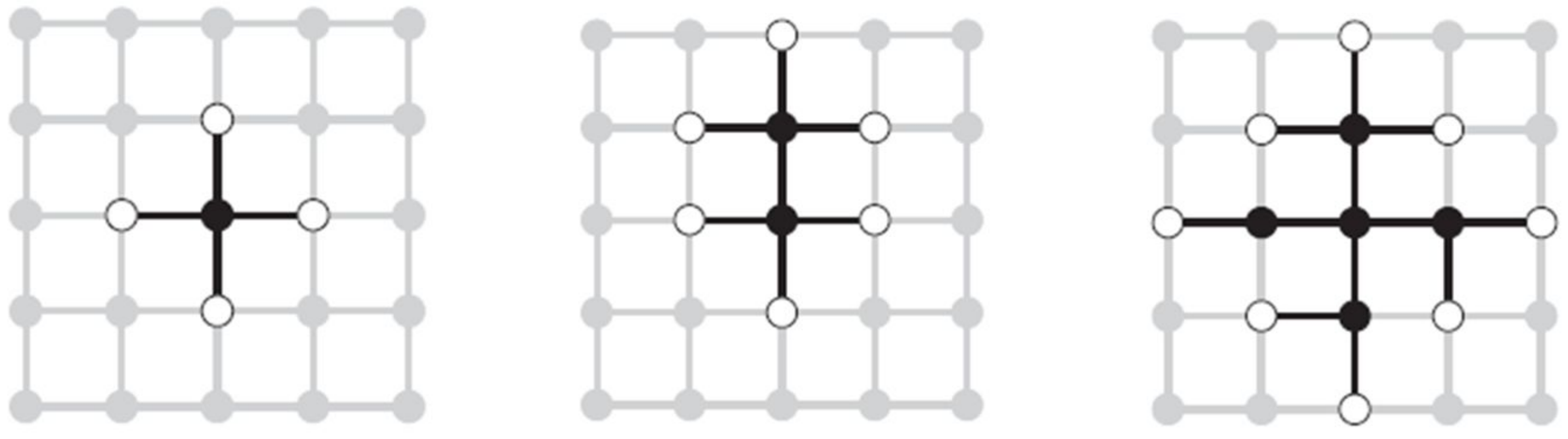


A Sequence of Search Trees



- A sequence of search trees generated by a graph search on the Romania problem.
- At each stage, each path is extended by one step.
- Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

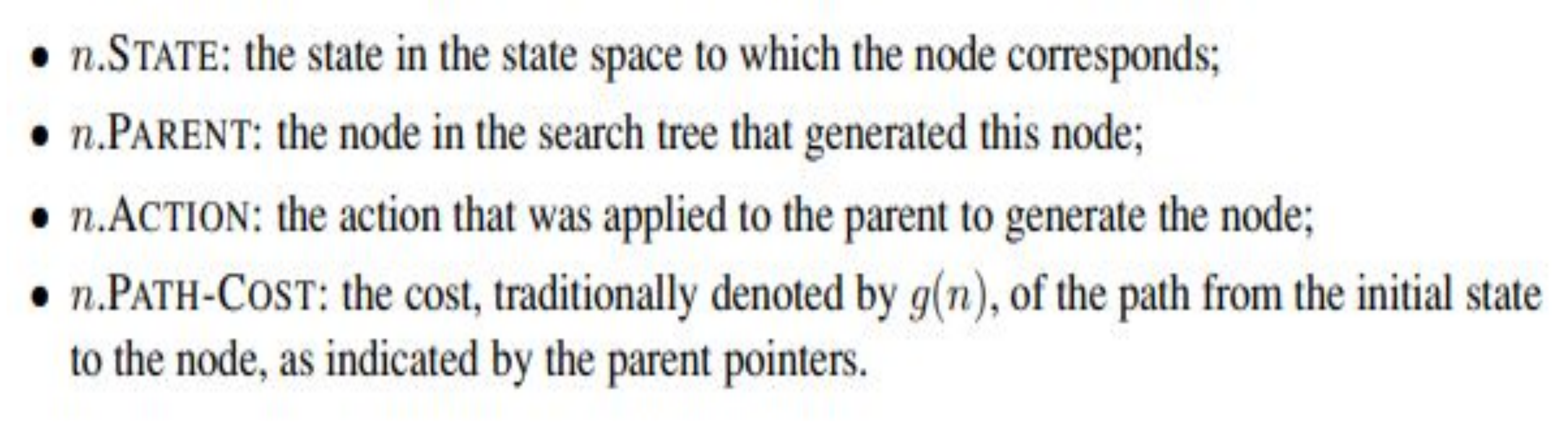
Separation Property of Graph-Search Algorithm


(a) (b) (c)

- The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem.
- The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes).
- In (a), just the root has been expanded.
- In (b), one leaf node has been expanded.
- In (c), the remaining successors of the root have been expanded in clockwise order.

Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed.
- For each node n of the tree, we have a structure that contains four components:



- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

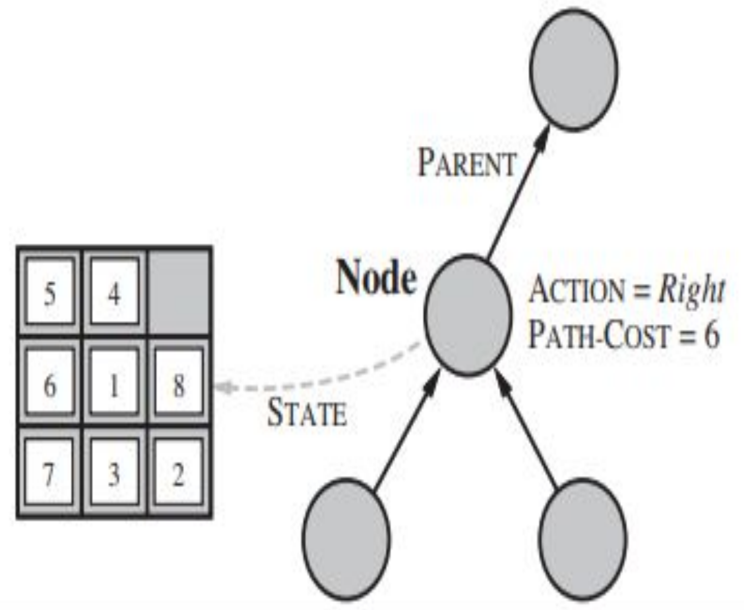
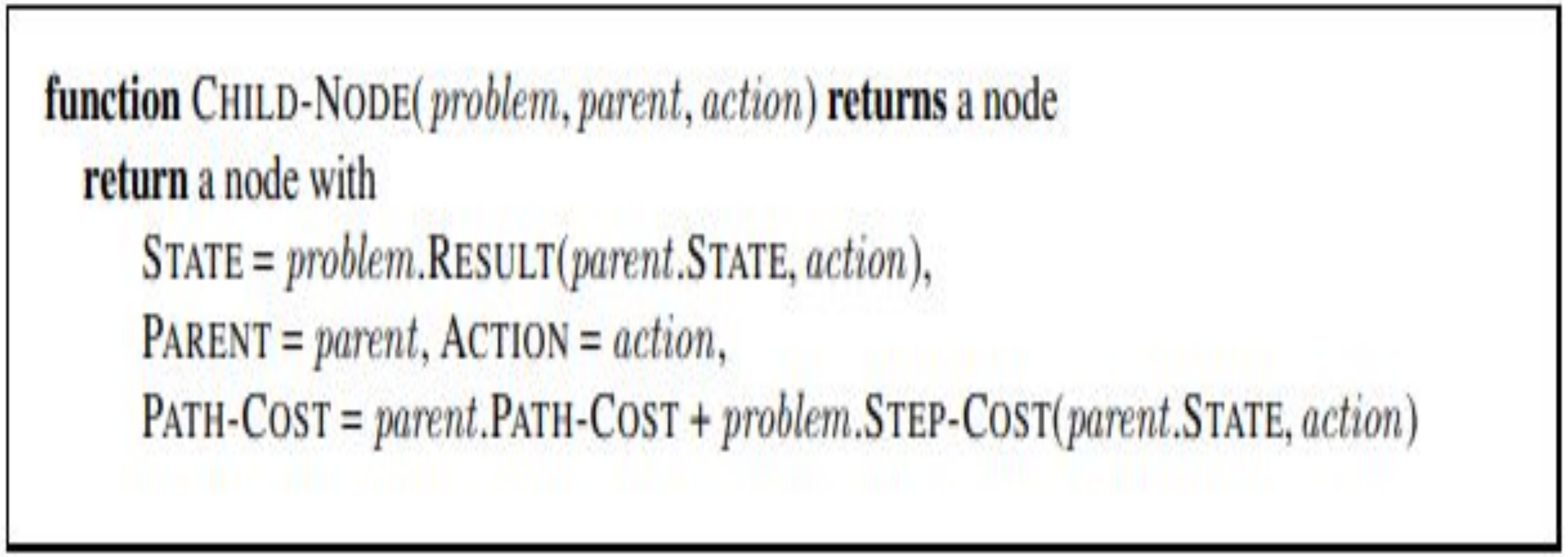


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

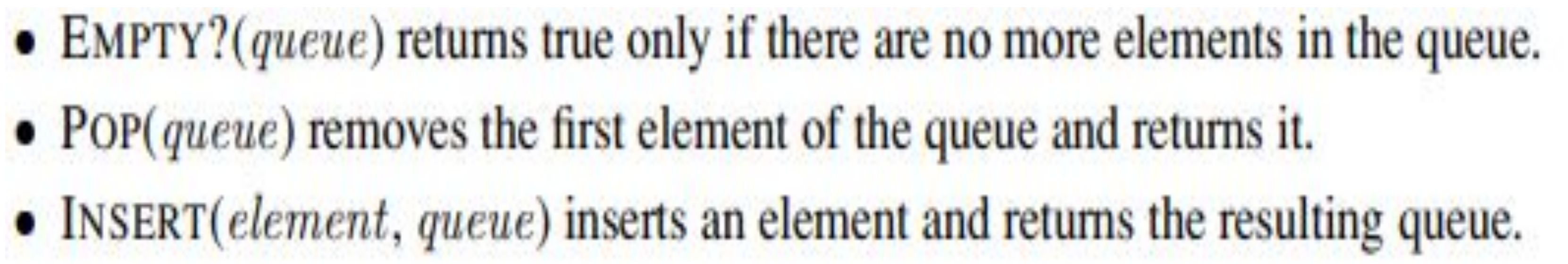
Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:



```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

- A node is a bookkeeping data structure used to represent the search tree.
- A state corresponds to a configuration of the world. Thus, nodes are on particular paths, as defined by PARENT pointers, whereas states are not.
- Furthermore, two different nodes can contain the same world state if that state is generated via two different search paths.
- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- The appropriate data structure for this is a queue.

The operations on a queue are as follows:



- EMPTY?(*queue*) returns true only if there are no more elements in the queue.
- POP(*queue*) removes the first element of the queue and returns it.
- INSERT(*element*, *queue*) inserts an element and returns the resulting queue.

Three common variants of queue are

◦ first-in, first-out or FIFO queue, which pops the oldest element of the queue;

◦ last-in, first-out or LIFO queue (also known as a stack), which pops the newest element of the queue

◦ priority queue, which pops the element of the queue with the highest priority according to some ordering function.

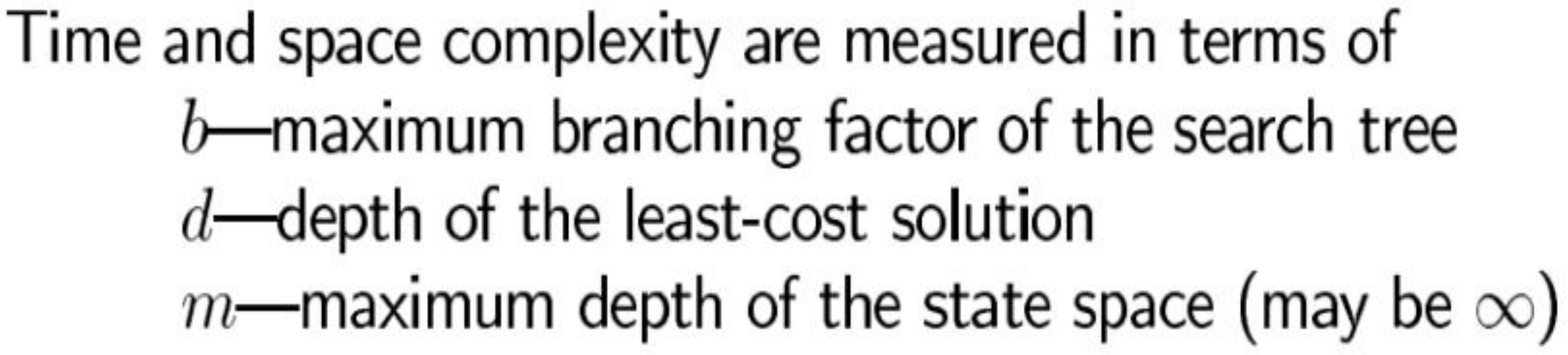
- The explored set can be implemented with a hash table to allow efficient checking for repeated states.
- With a good implementation, insertion and lookup can be done in roughly constant time no matter how many states are stored.
- One must take care to implement the hash table with the right notion of equality between states.

Measuring problem-solving performance

We can evaluate an algorithm's performance in four ways:

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- Optimality: Does the strategy find the optimal solution.
- Time complexity: How long does it take to find a solution?
- Space complexity: How much memory is needed to perform the search?

- Complexity is expressed in terms of three quantities: b, the branching factor or maximum number of successors of any node; d, the depth of the shallowest goal node and m, the maximum length of any path in the state space.


Time and space complexity are measured in terms of

b—maximum branching factor of the search tree

d—depth of the least-cost solution

m—maximum depth of the state space (may be ∞)

- Time is often measured in terms of the number of nodes generated during the search, and
- space in terms of the maximum number of nodes stored in memory.
- To assess the effectiveness of a search algorithm, we can use
  - - the **search cost**, which typically depends on the time complexity but can also include a term for memory usage; or
  - - the **total cost**, which combines the search cost and the path cost of the solution found.