

Knowledge Representation & Reasoning

COMP9016

Dr Ruairí O'Reilly
ruairi.oreilly@cit.ie

Problem Solving through Search

Problem solving agent

- >> **Problem-solving agent** – one kind of goal-based that uses **atomic** representations
- >> **Planning agents** – a different kind of goal-based agent that uses more advanced factored or structured representations of the world. (We'll see this later in the course)
- >> Defining **problems** and **solutions**
- >> **Uninformed** and **informed** search algorithms
- >> **NB:** Limitation – simplest kind of task environment – solution problem is a fixed sequence of actions

Goal & problem formulation

>> **Goal formulation** – Goals help organise behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider. Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

>> **Problem formulation** is the process of deciding what actions and states to consider, given a goal.

Example – Getting to Bucharest

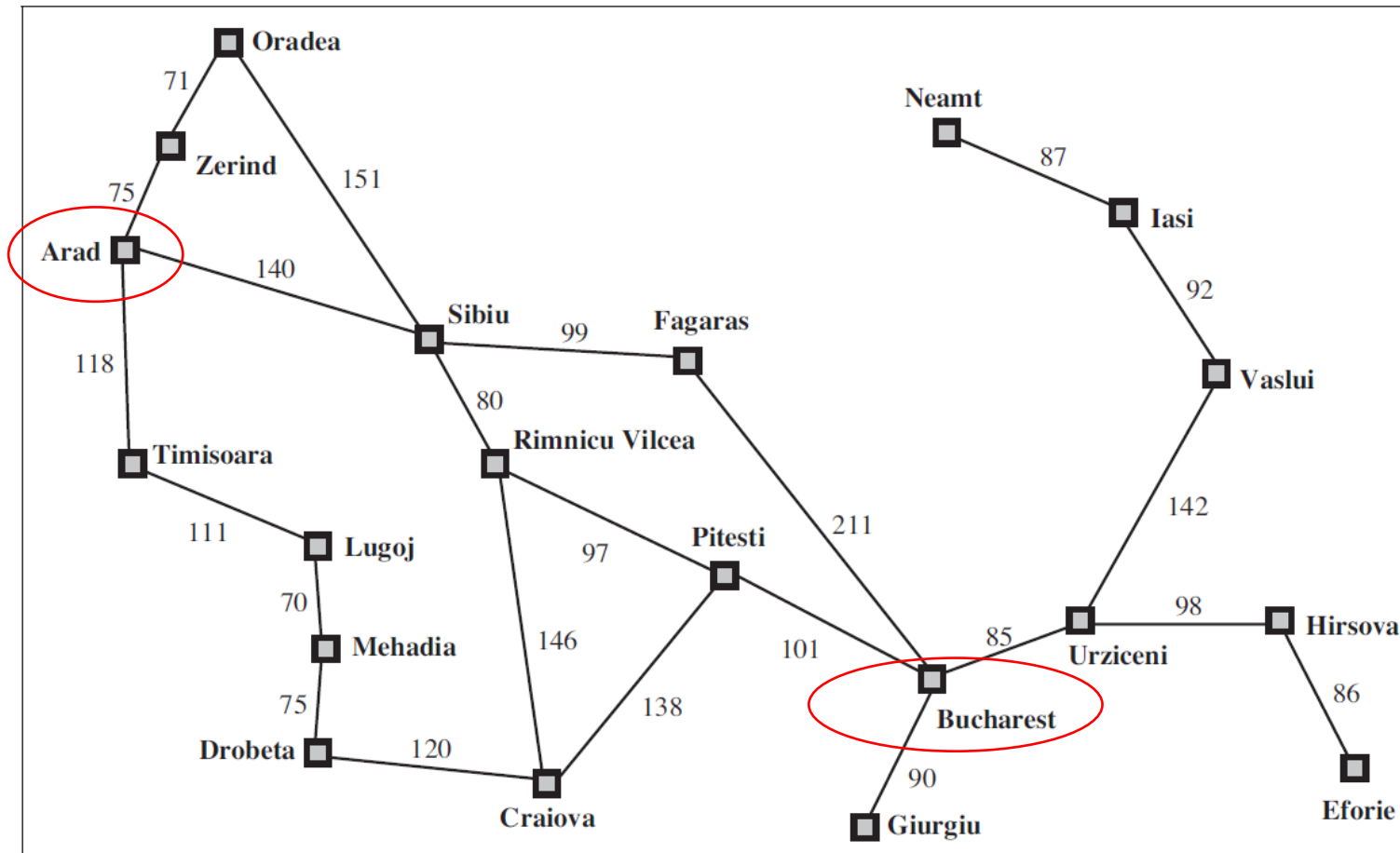


Figure 3.2 A simplified road map of part of Romania.

The Environment

>> It is necessary to “examine future actions”, to do so we have to be more specific about our environment:

>> **The environment is:**

- Observable
- Discrete
- Known
- Deterministic

Under these assumptions, the solution to any problem is a fixed sequence of actions. When is it not?

Search

>> Search - The process of looking for a sequence of actions that reaches the goal.

>> A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution phase**.

>> Thus, we have a simple “formulate, search, execute” design for the agent.

Formulate, search, execute

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action

persistent: *seq*, an action sequence, initially empty

state, some description of the current world state

goal, a goal, initially null

problem, a problem formulation

state \leftarrow UPDATE-STATE(*state*, *percept*)

if *seq* is empty **then**

goal \leftarrow FORMULATE-GOAL(*state*)

problem \leftarrow FORMULATE-PROBLEM(*state*, *goal*)

seq \leftarrow SEARCH(*problem*)

if *seq* = *failure* **then return** a null action

action \leftarrow FIRST(*seq*)

seq \leftarrow REST(*seq*)

return *action*

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Problem Formulation

Initial state e.g. $In(Arad)$

Actions – given state s , $ACTIONS(s)$ returns the set of actions that can be executed in s
e.g. from $In(Arad)$, the applicable actions are $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$

Transition model - $RESULT(s, a)$ returns the state that results from doing action a in state s
e.g. $RESULT(In(Arad), Go(Zerind)) = In(Zerind)$

State space & path – e.g. map of Romania (nodes are states, edges are actions)

- state-space graph: each road two driving actions
- path: a sequence of states connected by a sequence of actions.

Goal test e.g. $\{In(Bucharest)\}$.

Problem Formulation

A path cost - function that assigns a numeric cost to each path.

Step cost - we assume that the cost of a path can be described as the sum of the costs of the individual actions along the path.

The step cost of taking action a in state s to reach state s' is denoted by $c(s, a, s')$. The step costs for Romania are shown in Figure 3.2 as route distances. We assume that step costs are nonnegative.

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

Formulating problems

>> We proposed a formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost.

>> This formulation seems reasonable, but it is still a model—an abstract mathematical description—and not the real thing. Can you suggest additional real-world states not being considered?

>> Considerations that are irrelevant to the problem of finding a route to Bucharest are omitted. The process of removing detail from a **representation** is called **abstraction**.

>> Abstracting actions

>> Can we be more precise about defining the appropriate level of abstraction?

Toy problem – Vacuum world

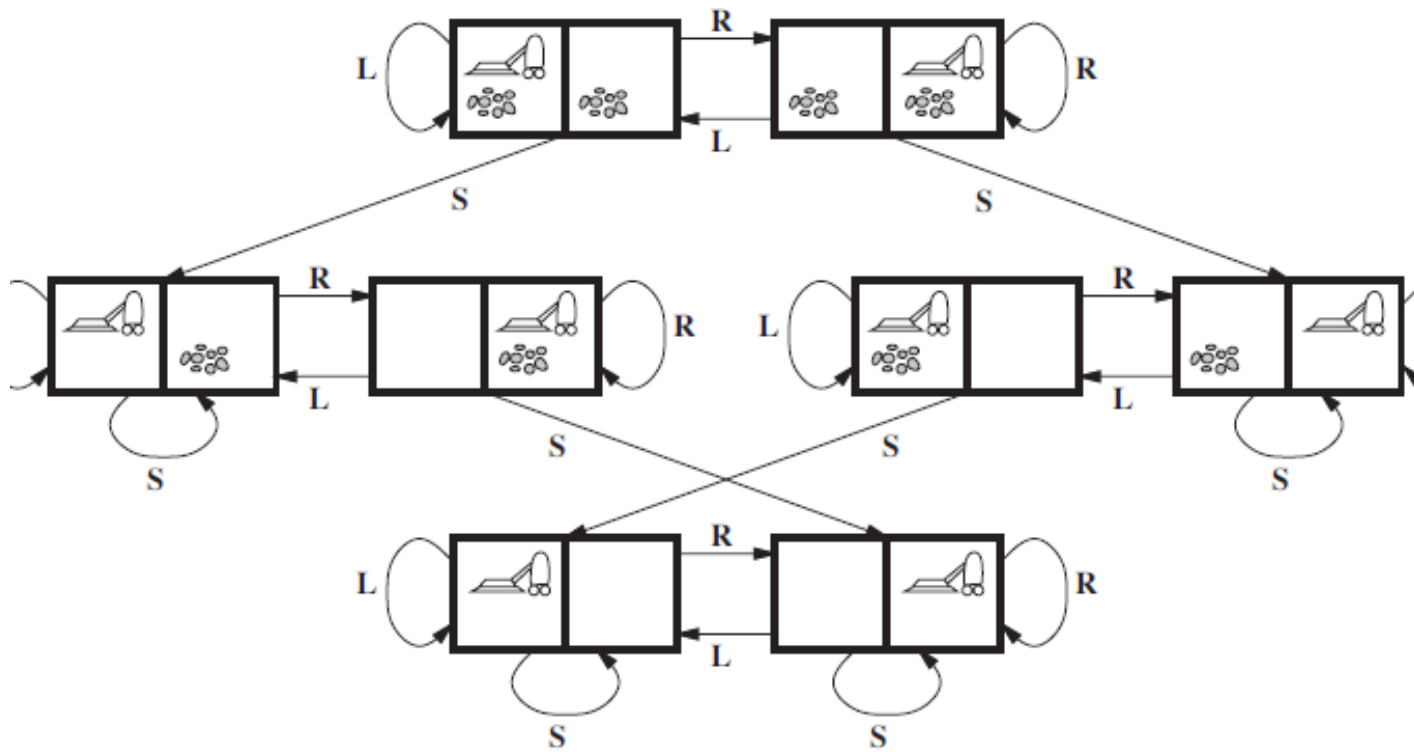


Figure 3.3 The state space for the vacuum world. Links denote actions: L = Left, R = Right, S = Suck.

- **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state:** Any state can be designated as the initial state.
- **Actions:** In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
- **Transition model:** The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in Figure 3.3.
- **Goal test:** This checks whether all the squares are clean.
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

Toy problem – 8-puzzle

7	2	4
5		6
8	3	1

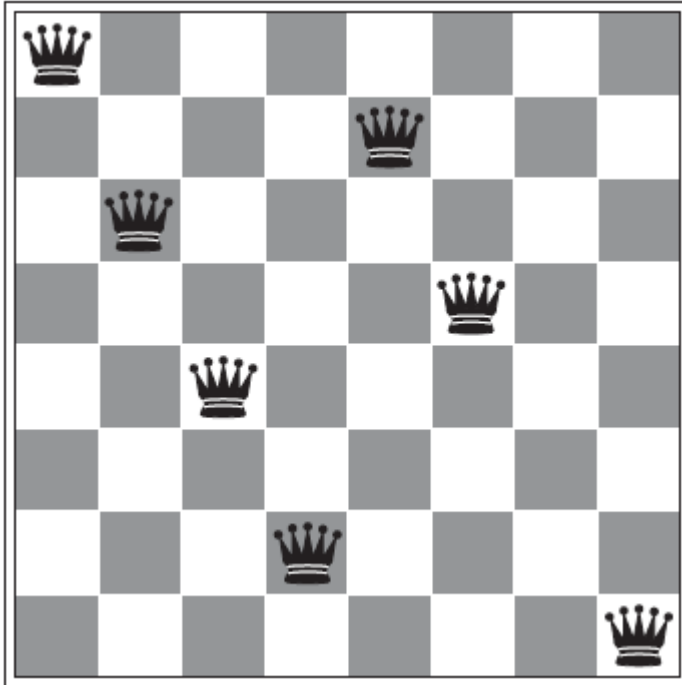
Start State

	1	2
3	4	5
6	7	8

Goal State

- **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares
- **Initial state:** Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- **Actions:** The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply Left to the start state, the resulting state has the 5 and the blank switched.
- **Goal test:** This checks whether the state matches the goal configuration shown in RHS (Other goal configurations are possible.)
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

Toy problem – 8-queens



Incremental formulation

Complete-state formulation

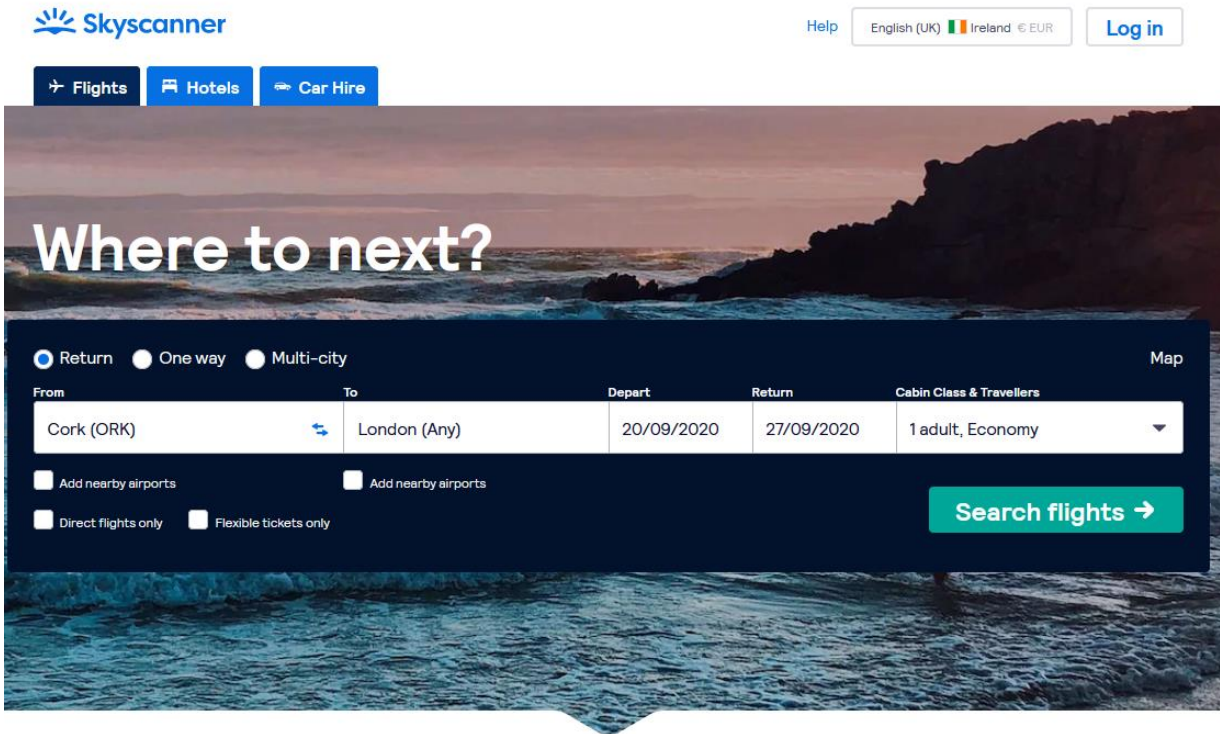
Incremental formulation v 0.2

- **States:** All possible arrangements of 0 to 8 queens on the board (0 ≤ n ≤ 8), one per column in the leftmost n columns, with no queen attacking another.
 - **Initial state:** No queens on the board.
 - **Actions:** Add a queen to any empty square.
 - **Initial state:** No queens on the board.
 - **Transition model:** Returns the board with a queen added to the specified square.
 - **Actions:** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
 - **Goal test:** 8 queens are on the board, none attacked.
 - **Transition model:** Returns the board with a queen added to the specified square.
 - **Goal test:** 8 queens are on the board, none attacked.
1. 8×10^{14} possible sequences to investigate.
A better formulation would prohibit placing a queen in any square that is already attacked:

This formulation reduces the 8-queens state space from 1.8×10^{14} to just 2,057, and solutions are easy to find.

On the other hand, for 100 queens the reduction is from roughly 10^{400} states to about 10^{52} states—a big improvement, but not enough to make the problem tractable.

Real world problem - route-finding e.g. Travel Planning



The screenshot shows the Skyscanner website with a search form for flights. The form includes fields for 'From' (Cork (ORK)), 'To' (London (Any)), 'Depart' (20/09/2020), 'Return' (27/09/2020), and 'Cabin Class & Travellers' (1 adult, Economy). There are also checkboxes for 'Add nearby airports', 'Direct flights only', and 'Flexible tickets only'. A 'Search flights' button is visible. The background of the form is a scenic image of a coastline at sunset.

- What can go wrong?
- What would a great system do?

- **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.

Real world problem - Very-large-scale integration (VLSI) layout

The screenshot shows a Glassdoor job search interface. At the top, the search bar contains 'vlsi engineer', with filters for 'Jobs', 'Ireland', and a 'Search' button. Below the search bar, there are tabs for 'Jobs', 'Companies', 'Salaries', and 'Interviews'. The main content area displays a list of job results. The first result is for 'IC Resources' (4.5 stars) for an 'ASIC Design Engineer' position in Cork, marked as 'New' and '3d' old. Other results include 'Qualcomm Incorporated' (4.0 stars) for a 'Digital ASIC R&D Engineer' in Cork, 'IT Search' (4.1 stars) for a 'Senior Embedded Engineer (C++) - R&D Medical Devices' in Dublin, 'Intel' (4.0 stars) for a 'Senior Deep Learning Verification Engineer' in Leixlip, and 'Movidius' (3.3 stars) for a 'Senior Deep Learning Verification Engineer' in Leixlip. On the right side of the first result, there is a detailed view of the 'ASIC Design Engineer' job at 'IC Resources' (4.5 stars). This view includes an 'Apply Now' button, a 'Save' button, and a 'New' badge. Below the job title, it says 'Be one of the first to Apply!'. The job details section includes a table with columns for 'Job', 'Company', 'Rating', and 'Salary'. The salary information states 'Salary: Stock, excellent salary and benefits!' and 'Job Type: Permanent'. The job description mentions 'ASIC Design Engineer!! R&D!! Semiconductor giant!' and describes the role as a fantastic opportunity to join one of the world's biggest semiconductor companies. It also lists the responsibilities of the ASIC Design Engineer, including addressing challenges with Performance, Power, and Area (PPA) scaling tradeoffs. The minimum qualifications listed are a Bachelor's degree in Science, Engineering, or related field, and preferred qualifications include scripting with Python, TCL, and/or C++, basic knowledge in digital VLSI implementation, hands-on experience with EDA tools, and 2+ years of experience in software development.

glassdoor

vlsi engineer

Jobs

Ireland

Search

Jobs Companies Salaries Interviews

For Employers Post Jobs

All Job Types

Posted Any Time

25 Miles

All Cities

More

IC Resources 4.5 ★

ASIC Design Engineer

Cork

New 3d

Qualcomm Incorporated 4.0 ★

Digital ASIC R&D Engineer, QCT, Cork, Ireland

Cork

19d

IT Search 4.1 ★

Senior Embedded Engineer (C++) - R&D Medical Devices

Dublin

Easy Apply 20d

Intel 4.0 ★

Senior Deep Learning Verification Engineer

Leixlip

New 3d

Movidius 3.3 ★

Senior Deep Learning Verification Engineer

Leixlip

24d

Page 1 of 1

roreilly@gmail.com

Create Job Alert

People Also Searched

IC Resources 4.5 ★

ASIC Design Engineer

Cork

Apply Now Save

New Be one of the first to Apply!

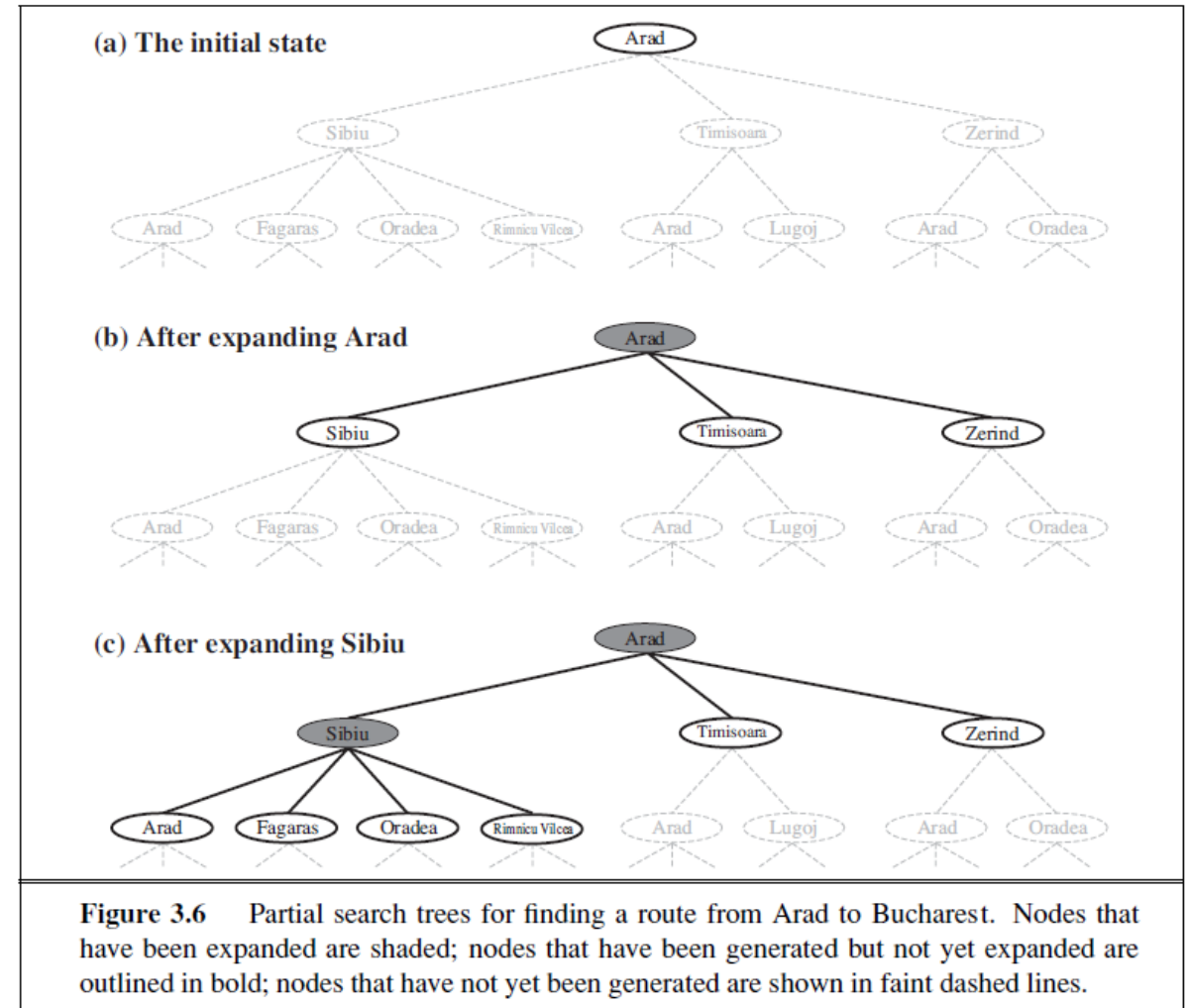
Job	Company	Rating	Salary
Salary: Stock, excellent salary and benefits!			
Job Type: Permanent			
ASIC Design Engineer!! R&D!! Semiconductor giant!			
This is a fantastic opportunity to join one of the worlds biggest Semiconductor companies in their European HQ! My clients Design Technology Team is actively seeking candidates for VLSI R&D positions. You will be part of the design methodology team that develops innovative solutions for chip implementation flow.			
As the ASIC Design Engineer, you will play a vital role in addressing challenges with Performance, Power, and Area (PPA) scaling tradeoffs to qualify technology entitlement of advance process nodes. You will be responsible for convergence of complex designs using best-in-class tools, flows & methodologies to optimise PPA.			
Minimum Qualifications			
Bachelor's degree in Science, Engineering, or related field.			
Preferred Qualifications			
Scripting with Python, TCL and/or C++			
Basic knowledge in digital VLSI implementation (netlist to GDS)			
Hands on experience with EDA tools (Primetime, ICC2, Innovus, Tempus, etc.)			
2+ years of experience in software development			

- A 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.

ASIC: Application-specific integrated circuit

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.
- 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.



Tree Search

- **Search Strategy** – how to choose which state to expand
- 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.

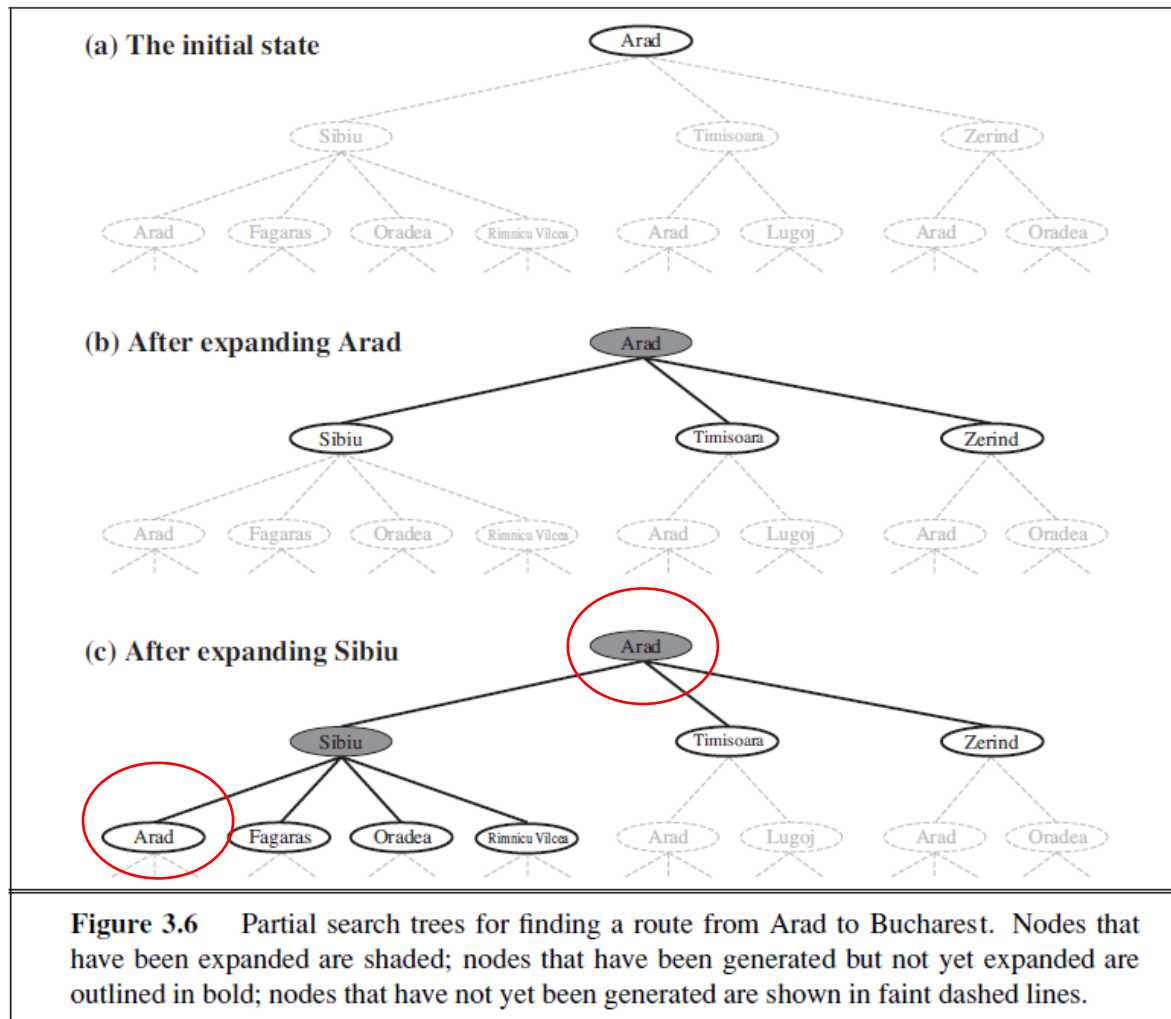
```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding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

```
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
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Redundant Paths

- 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)?



E.g. Rectangular Grid

- Routing on a rectangular grid is a particularly important example in computer games. 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.
- “*algorithms that forget their history are doomed to repeat it*”. The way to avoid exploring redundant paths is to remember where one has been. See GRAPH-SEARCH
- The search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph, as shown in Figure 3.8

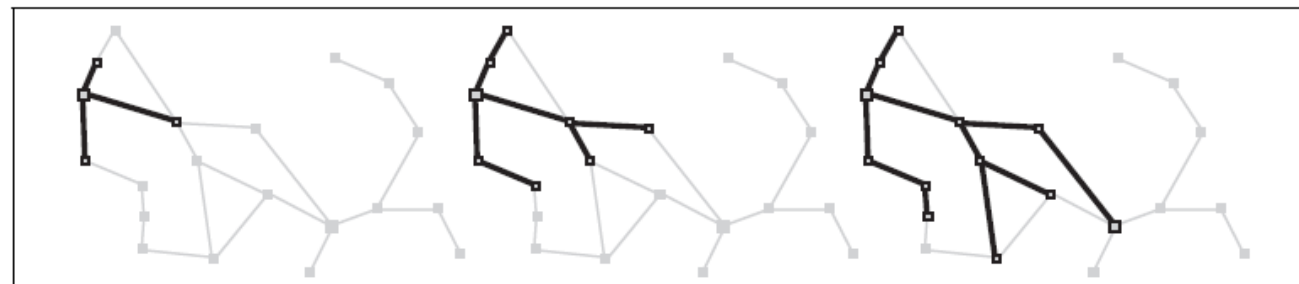


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

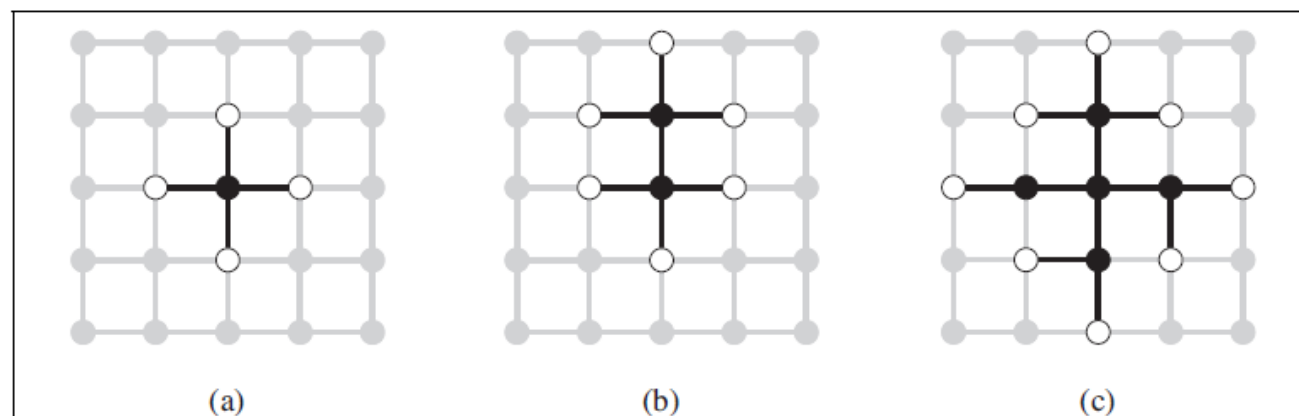


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

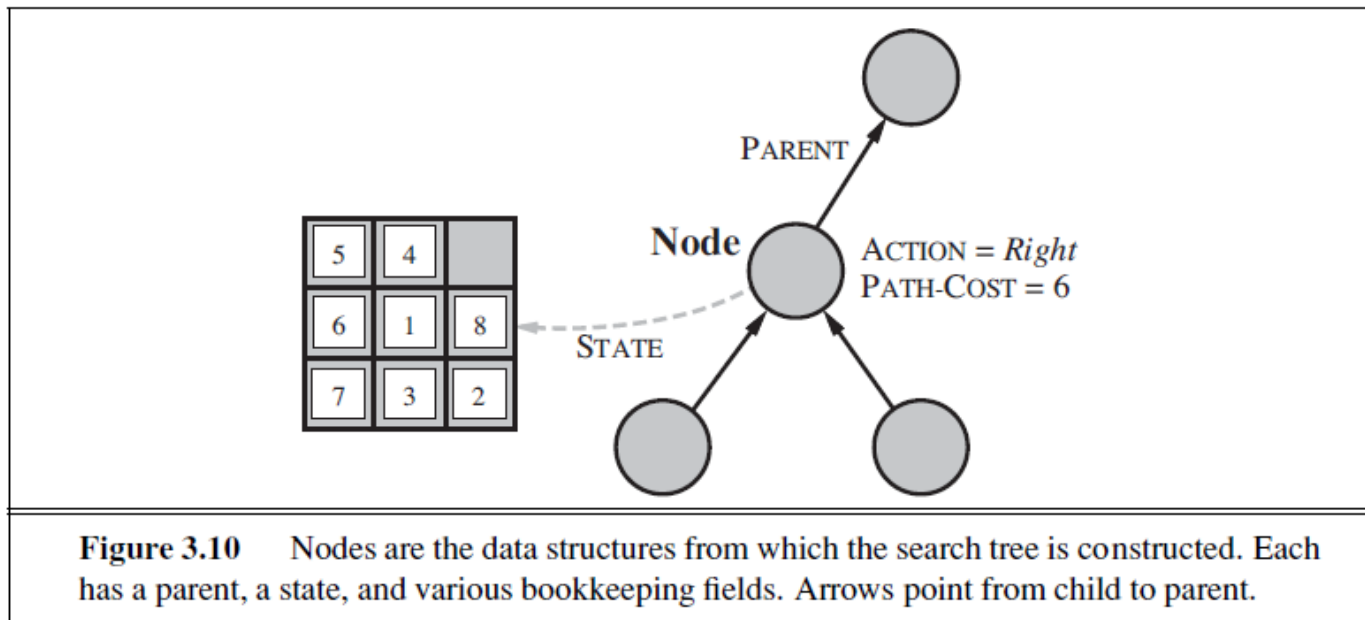
Infrastructure for search algorithms

>> 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.

Node Data Structure

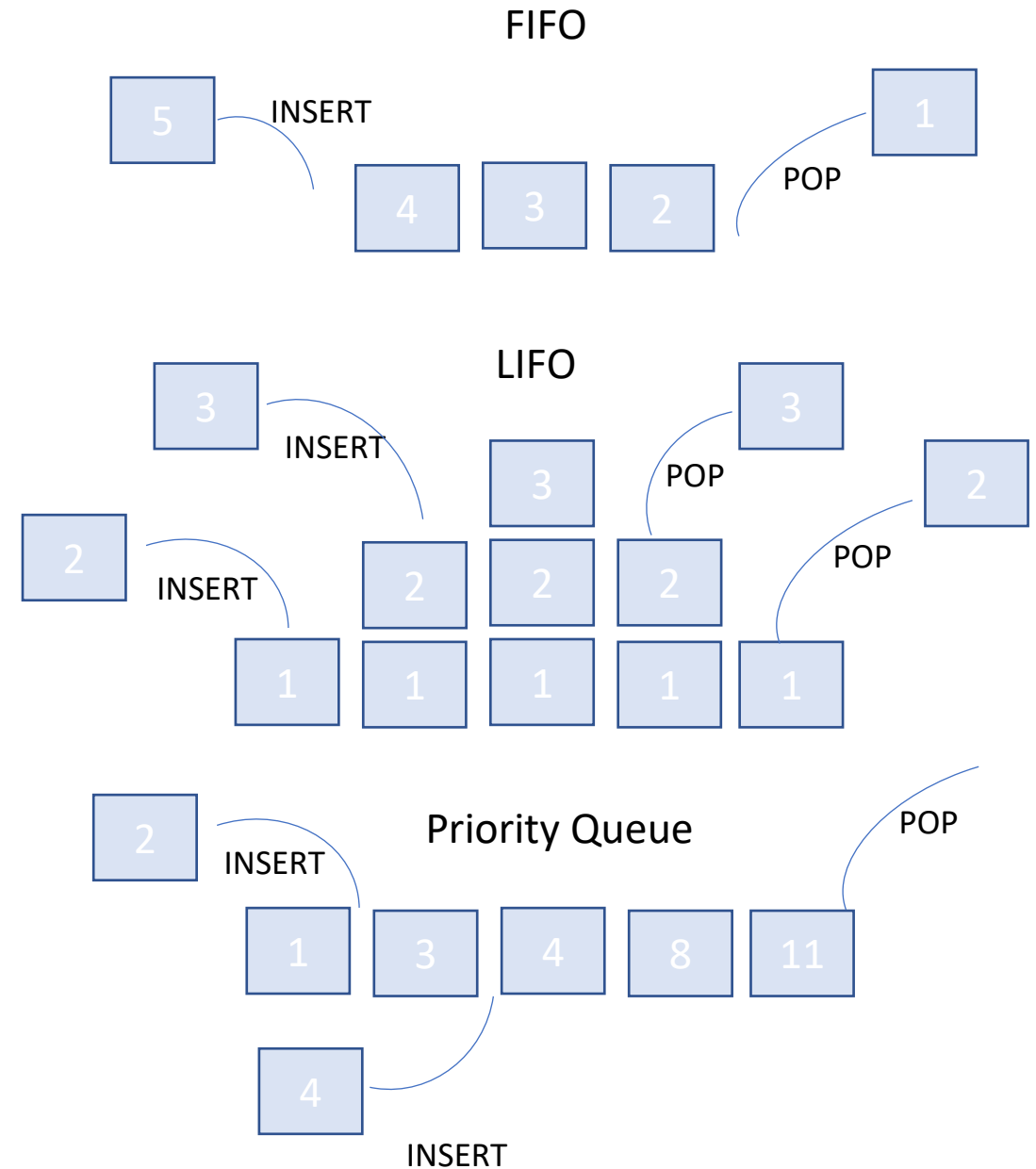
- Function CHILD-NODE takes a parent node and an action and returns the resulting child node.
- Distinguishing between nodes and states: A node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world.
- Now that we have nodes, we need somewhere to put them: **QUEUE**



```
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)
```

Queue Data Structure

- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand QUEUE 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.
- LIFO, FIFO & Priority queue
- Canonical form
 $\{\text{Bucharest, Urziceni, Vaslui}\} = \{\text{Urziceni, Vaslui, Bucharest}\}$



Measuring problem-solving performance

>> Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. 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?

Time and space complexity are always considered with respect to some measure of the problem difficulty.

The state space graph: $|V| + |E|$

Measuring problem-solving performance

>> In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite.

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 (i.e., the number of steps along the path from the root).
- m , the maximum length of any path in the state space.

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.

Uninformed search strategies

>> This means strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded.

- Breadth first search
- Uniform cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Breadth-first search

- A simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- General graph-search, shallowest unexpanded node chosen, FIFO queue for the frontier.

```

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier) /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
  
```

Figure 3.11 Breadth-first search on a graph.

How does BFS rate according to the four criteria?

>> It is complete

>> Time complexity:

Imagine searching a uniform tree where every state has b successors.

$$b + b^2 + b^3 + \dots + b^d = O(bd)$$

>> Space complexity:

$$\text{Explored set: } O(b^{(d-1)}) \quad \text{Frontier: } O(b^d)$$

$$\Rightarrow \text{Space complexity} = O(b^d)$$

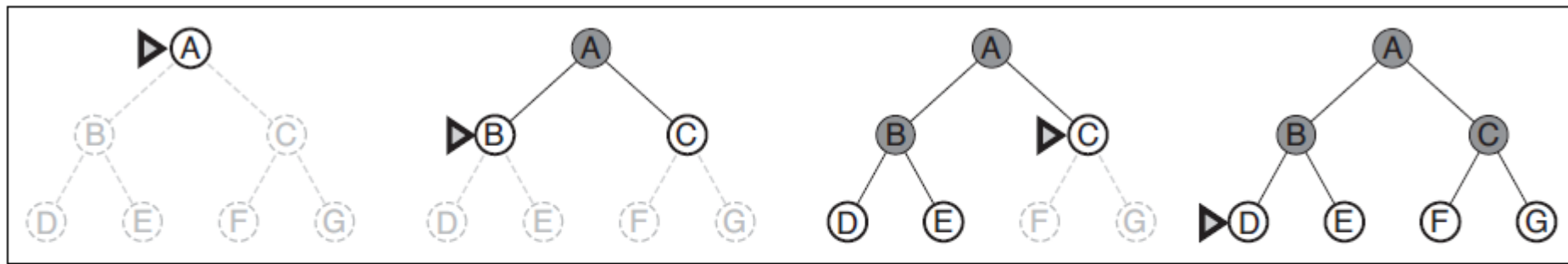


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Exponential complexity bound is a problem

>> The following table assumes a branching factor $b = 10$, 1 million nodes can be generated per second and that a node requires 1000 bytes of storage.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

>> The memory requirements are a bigger problem for breadth-first search than is the execution time.

>> Time is still a major factor In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

Uniform-cost search

- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost $g(n)$. This is done by storing the frontier as a priority queue ordered by g .
- Goal test is applied to a node when it is selected for expansion rather than when it is first generated. The reason is that the first goal node that is generated may be on a suboptimal path
- The second difference is that a test is added in case a better path is found to a node currently on the frontier.

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node \leftarrow a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

frontier \leftarrow a priority queue ordered by PATH-COST, with *node* as the only element

explored \leftarrow an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node \leftarrow POP(*frontier*) /* chooses the lowest-cost node in *frontier* */

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

 add *node*.STATE to *explored*

for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

child \leftarrow CHILD-NODE(*problem*, *node*, *action*)

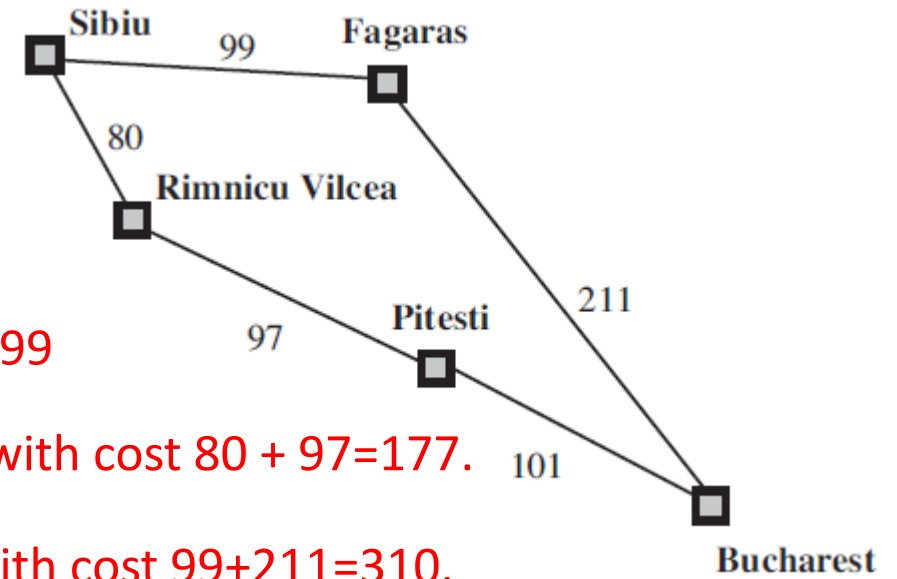
if *child*.STATE is not in *explored* or *frontier* **then**

frontier \leftarrow INSERT(*child*, *frontier*)

else if *child*.STATE is in *frontier* with higher PATH-COST **then**

 replace that *frontier* node with *child*

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.



>> Problem is to get from Sibiu to Bucharest.

>> Successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99

>> The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost $80 + 97 = 177$.

>> Least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost $99 + 211 = 310$.

>> A goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost $80 + 97 + 101 = 278$.

Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

Depth-first search

- Expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.
- Is an instance of the graph-search; whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.
- Graph-search or tree-search version?

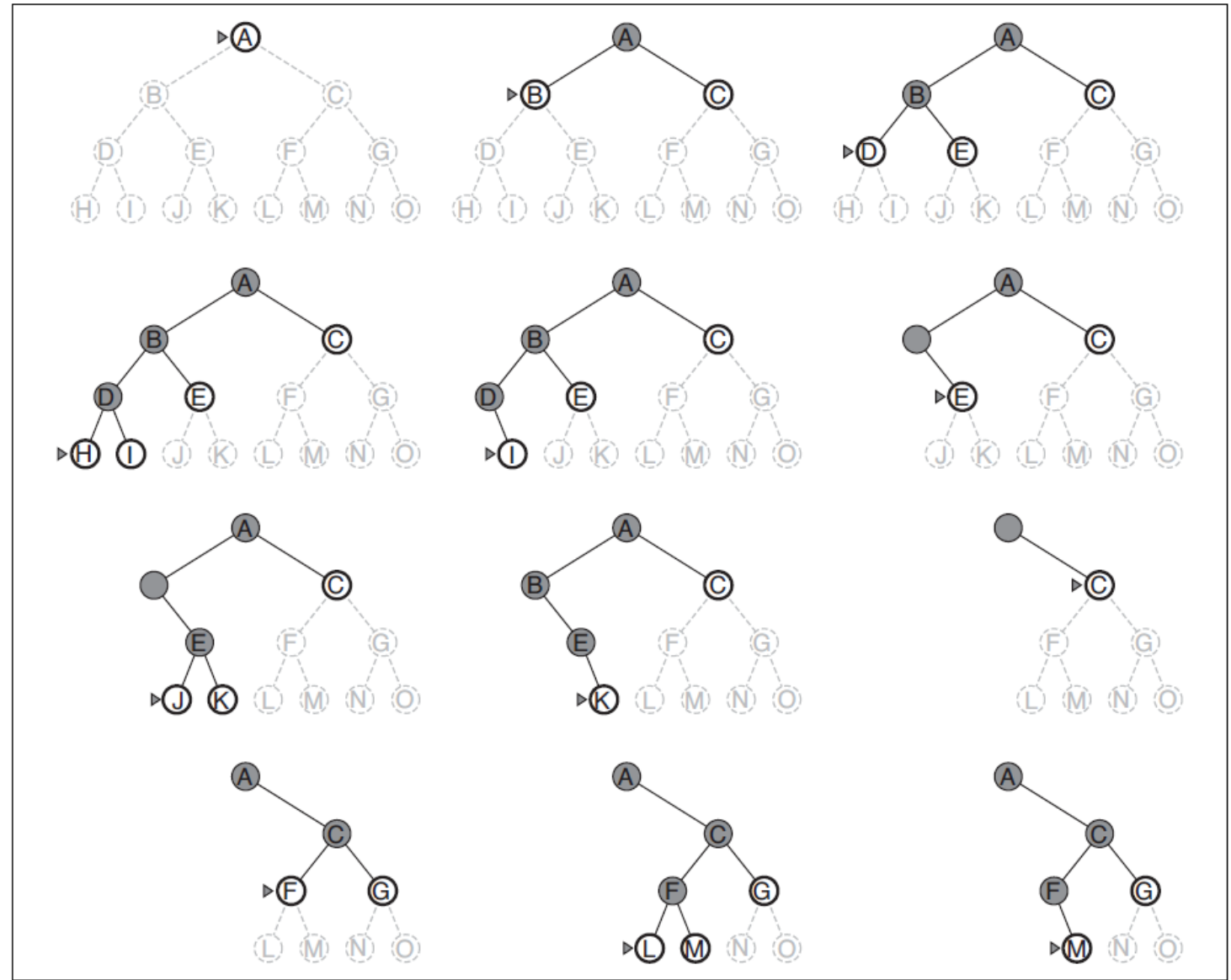


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and *M* is the only goal node.

Depth-limited search

- The failure of DFS in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit l . That is, nodes at depth l are treated as if they have no successors.
- additional source of incompleteness if we choose $l < d$
- What if $l > d$?
- *Time complexity:* $O(b^l)$ *Space complexity:* $O(bl)$. DFS can be viewed as a special case of depth-limited search with $l = \infty$

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred?  $\leftarrow$  false
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true
      else if result  $\neq$  failure then return result
    if cutoff_occurred? then return cutoff else return failure
  
```

Figure 3.17 A recursive implementation of depth-limited tree search.

Iterative deepening depth-first search

- Iterative deepening search is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure  
  for depth = 0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
    if result  $\neq$  cutoff then return result
```

Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

Iterative deepening depth-first search

- Is it wasteful? Total num of nodes in worst-case:

$$N(IDS) = (d)b + (d - 1)b^2 + \dots + (1)b^d$$

which gives a time complexity of $O(b^d)$ —asymptotically the same as breadth-first search. There is some extra cost for generating the upper levels multiple times, but it is not large. For example, if $b = 10$ and $d = 5$, the numbers are

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

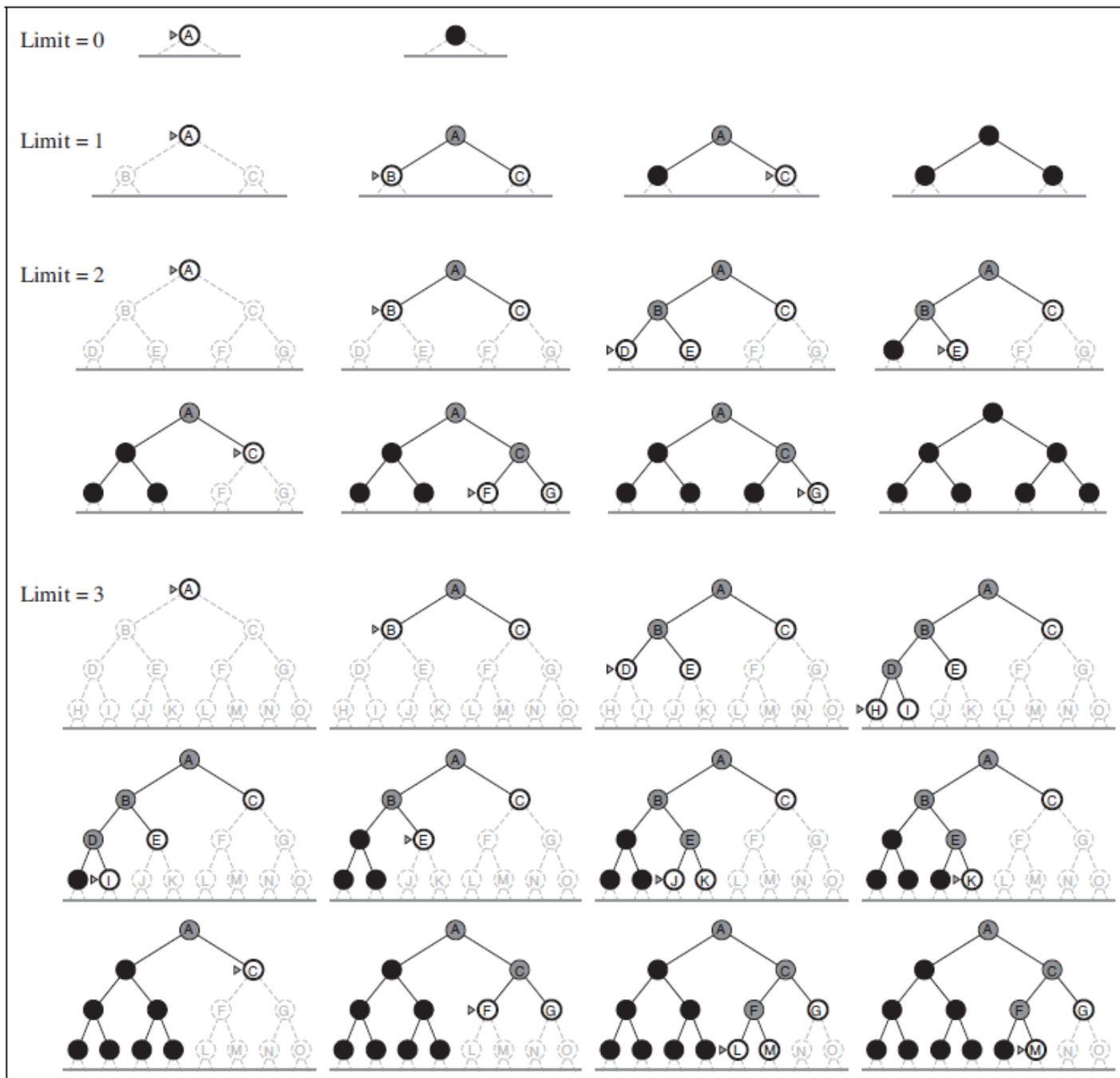
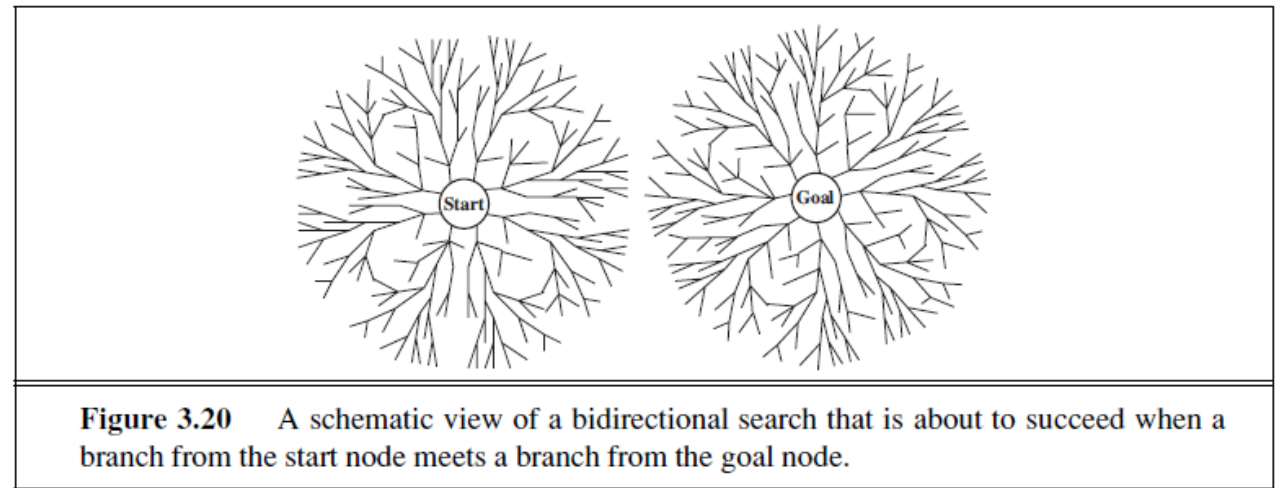


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Bidirectional search



- Two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.
- The motivation is that $b^{(d/2)} + b^{(d/2)}$ is much less than b^d , or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

Comparing uninformed search strategies

- This comparison is for tree-search versions. For graph searches, the main differences are that depth-first search is complete for finite state spaces and that the space and time complexities are bounded by the size of the state space.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b^l)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^a complete if b is finite; ^b complete if step costs $\geq \epsilon$ for positive ϵ ; ^c optimal if step costs are all identical; ^d if both directions use breadth-first search.

Informed (heuristic) search strategies

>> An informed search strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.

>> The general approach we consider is called best-first search. Best-first search is an instance of the general *TREE-SEARCH* or *GRAPH-SEARCH* algorithm in which a node is selected for expansion based on an evaluation function, $f(n)$. The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search, except for the use of f instead of g to order the priority queue.

>> The choice of f determines the search strategy.

Informed (heuristic) search strategies

>> Most best-first algorithms include as a component of f a **heuristic function**, denoted $h(n)$:

$h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.

>> Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. For now, we consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then $h(n)=0$.

Summary

- TO BE COMPLETED

References

[1] Russell, S. and Norvig, P., 2002. Artificial intelligence: a modern approach – Solving Problems by Searching, Chapter 3