Agents and search

How to interact with the environment and how to take decisions

From: Russell – Norvig «Artificial intelligence»

Prentice Hall

- The structure of Agents
- Solving problems by searching: uninformed (blind) search
- Informed search

Intelligent agents

What is an agent?

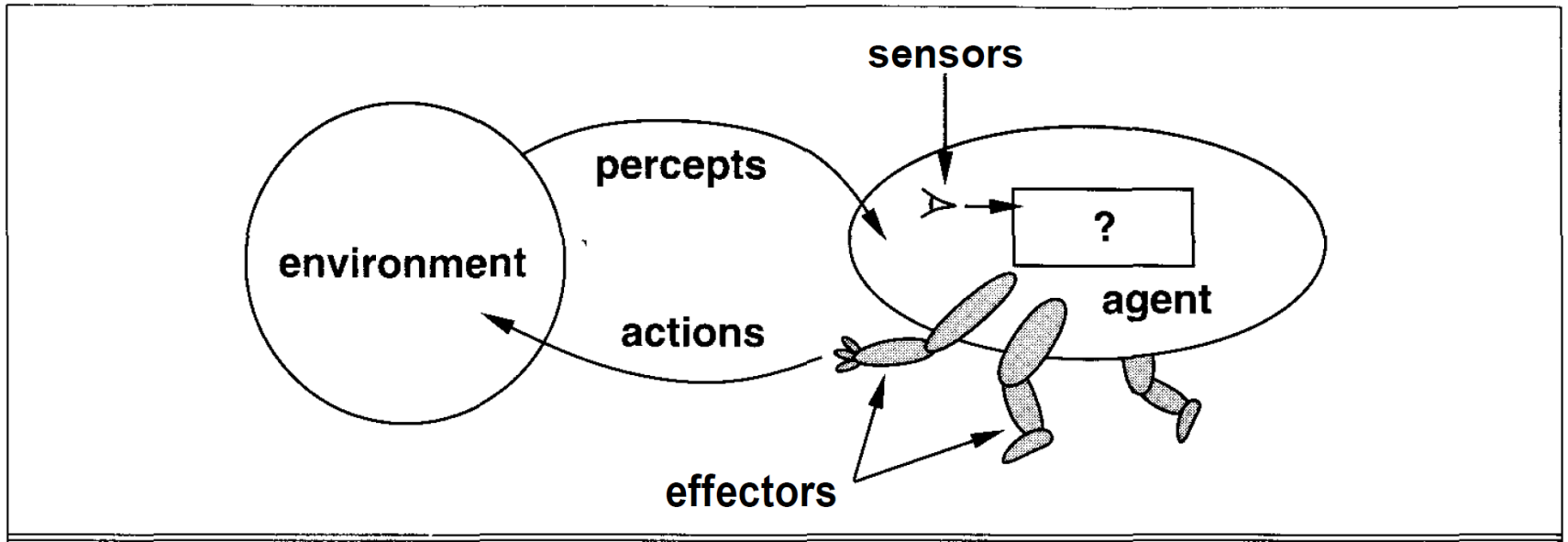


Figure 2.1 Agents interact with environments through sensors and effectors.

Ideal rational agent

For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

1. Performance measure
2. Agent’s a priori knowledge
3. Agent’s percept sequence
4. Actions the agent can really perform

Table driven agent

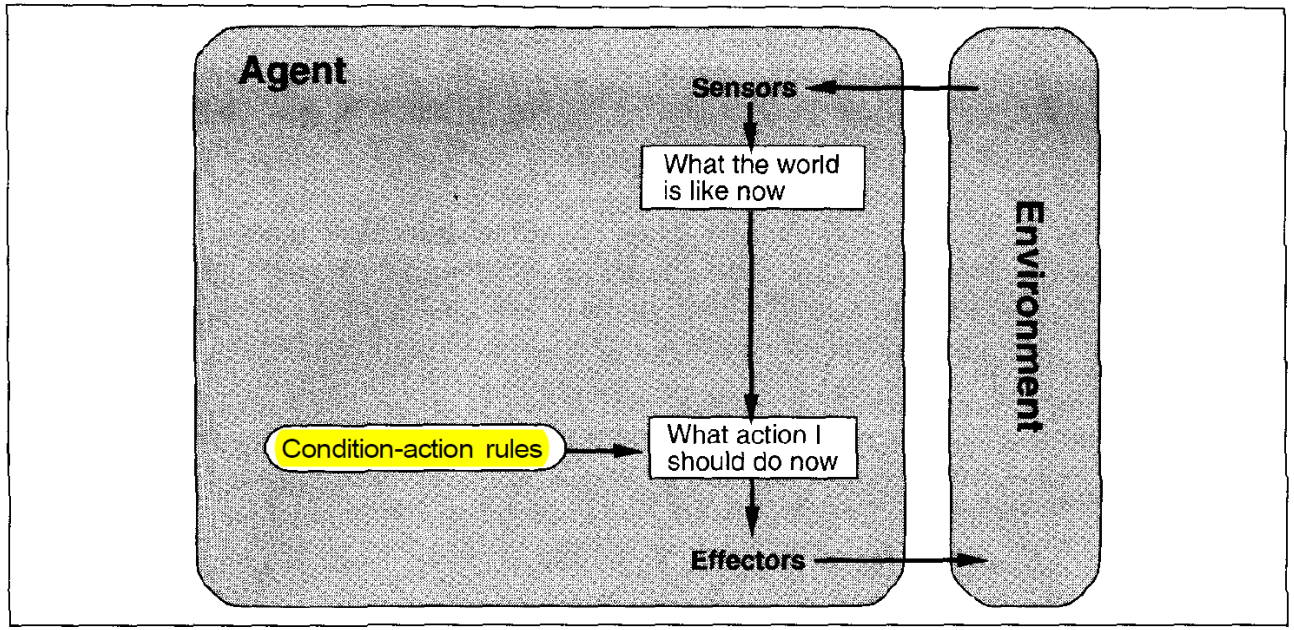
```
function TABLE-DRIVEN-AGENT(percept) returns action
  static: percepts, a sequence, initially empty
          table, a table, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

The table of the logaritms!

If P is the set of possible percepts and T the «lifetime» the lookup table will contain $\sum_{t=1}^{T} P^t$ entries!

Lookup table for chess contains 10^{150} entries!

Simple reflex agent



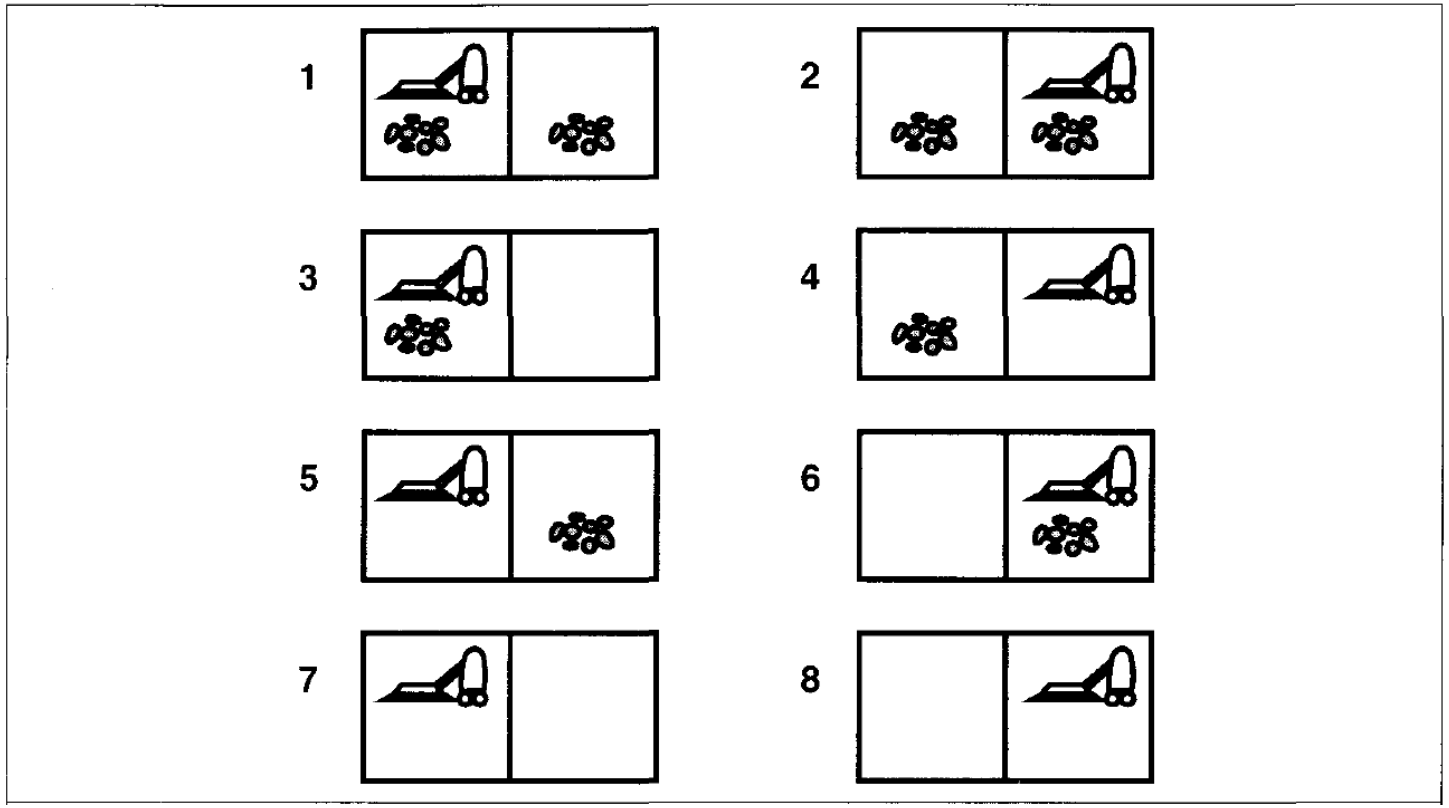

```
function SIMPLE-REFLEX-AGENT(percept) returns action
  static: rules, a set of condition-action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  return action
```

Ignore the percept history:

$\Sigma_{t=1}^{T} P^t \rightarrow P$

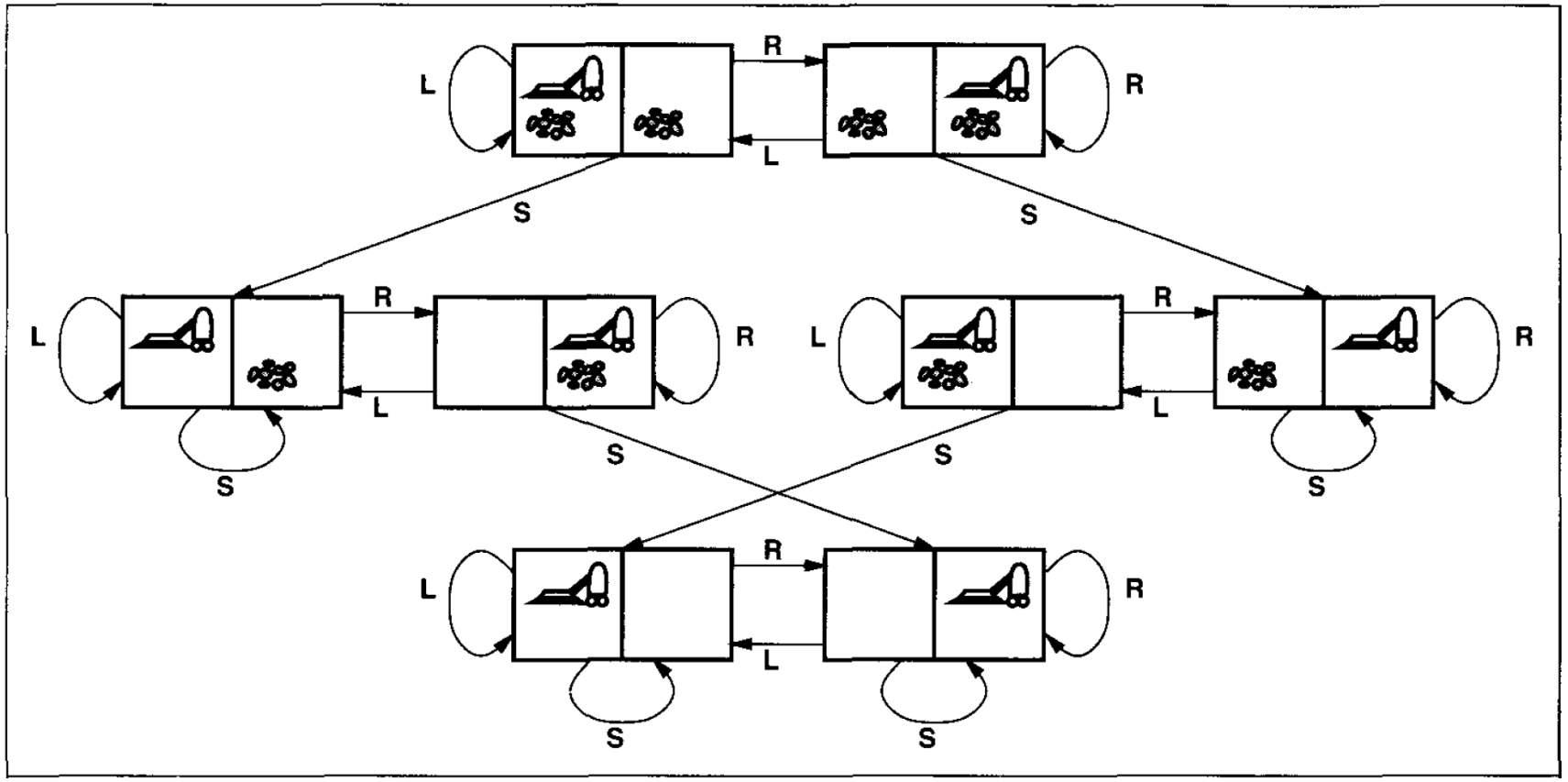
The environment need to be fully observable

The 8 possible states of a simplified vacuum cleaner



Number of states for n rooms either clean or dirty: $2_{c,d}{}^{n} \times n$

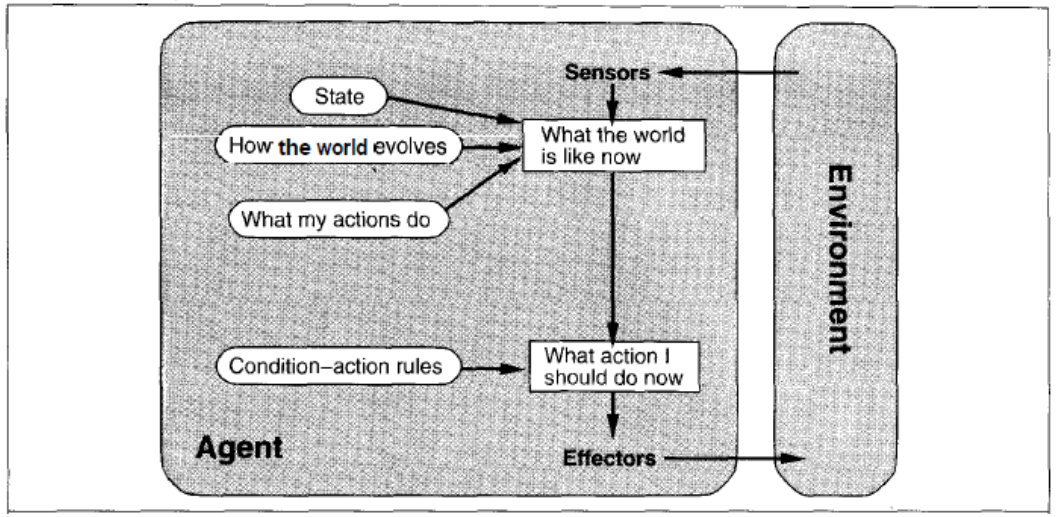
Actions on the state space of the vacuum cleaner



If the vacuum cleaner knows its position and if the room is clean/dirty a simple reflex agent can be implemented

Incomplete information? (sensor broken) → random action to escape the loop

Model-based reflex agent



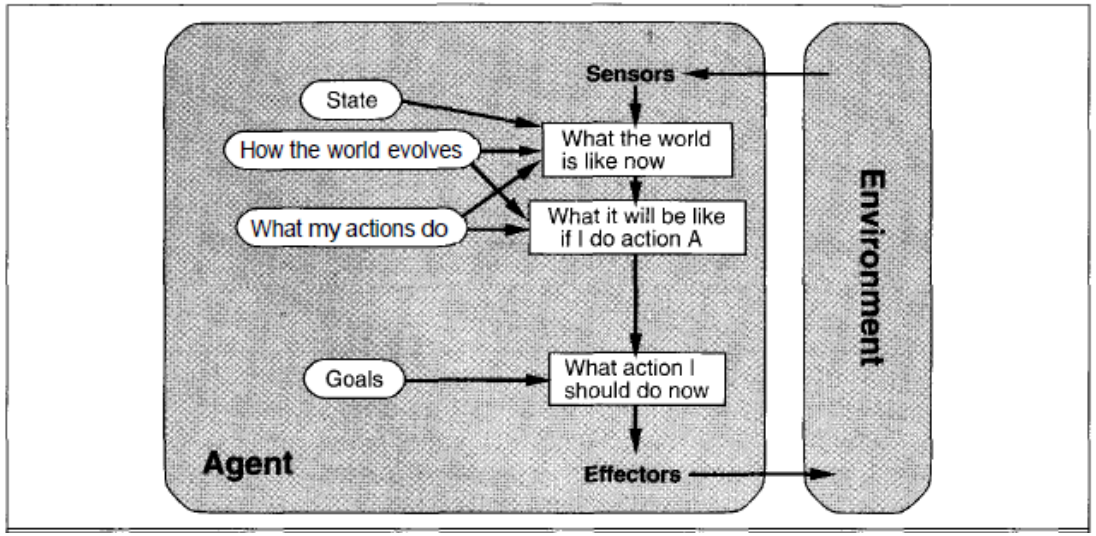

```
function REFLEX-AGENT-WITH-STATE(percept) returns action
  static: state, a description of the current world state
          rules, a set of condition-action rules

  state ← UPDATE-STATE(state, percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  state ← UPDATE-STATE(state, action)
  return action
```

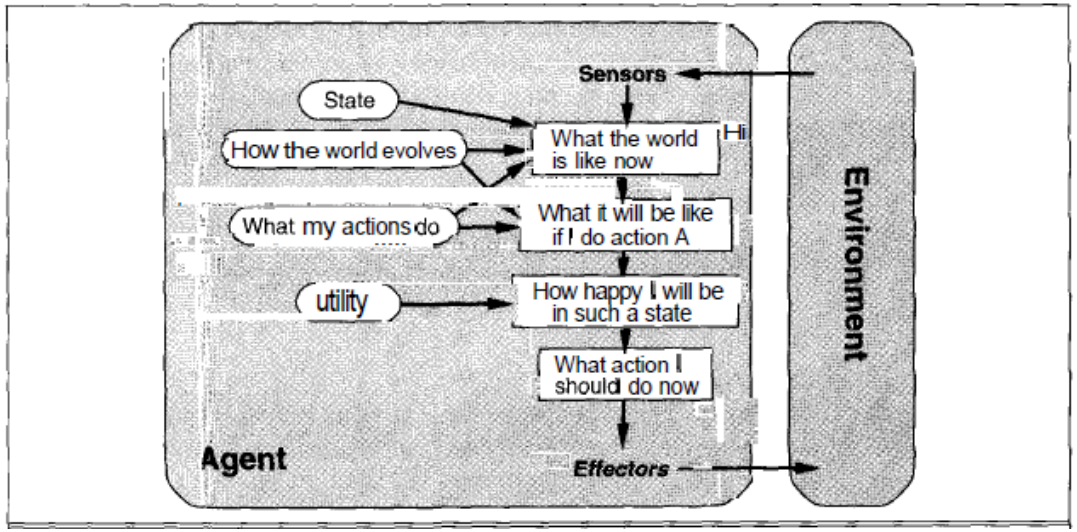
Keeps track of the part of the world it can't see now.

Driver: keeps track of the previous information about lights: allows to understand if the car in front is breaking.

Contains a model of the world: how the world works!

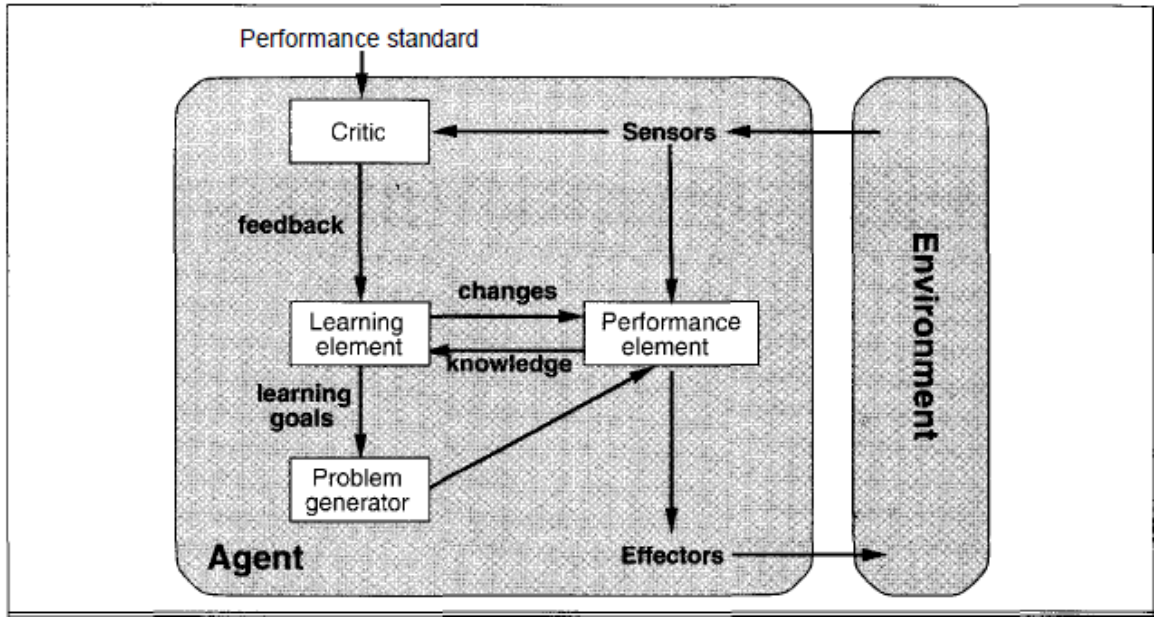
Goal-based and utility-based agents





An agent is more flexible if it knows what it has to achieve. Simple situations can be described in terms of a goal to be reached. More complex situations need the introduction of a utility function (describing the degree of happiness) that needs to be maximized. For instance, to modify an agent taxi-driver is simple IF it knows its goal, i.e. the destination. Moreover it can be useful to include also other parameters as safety, time needed, cost which enter the utility function.

Learning agents



Performance element: what discussed before
Learning element: responsible for the improvements
Critic: tells how well the agent is doing (must be fixed, potentially outside of the agent)
Problem generator: suggests actions leading to new informative experiences.

The learning element can make changes to any of the «knowledge» components of the agent.

- Observation of pairs of successive states → learn how the world evolves
- Observation of the result of an action → learn «what my actions do»
- …

Examples of agents

Agent Type	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Refinery controller	Temperature, pressure readings	Open, close valves; adjust temperature	Maximize purity, yield, safety	Refinery
Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

Properties of environments

Environments come in several flavors. The principal distinctions to be made are as follows:

ACCESSIBLE

- **Accessible** vs. **inaccessible.**
  If an agent's sensory apparatus gives it access to the complete state of the environment, then we say that the environment is accessible to that agent. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action. An accessible environment is convenient because the agent need not maintain any internal state to keep track of the world.

DETERMINISTIC

- **Deterministic** vs. **nondeterministic.**
  If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic. In principle, an agent need not worry about uncertainty in an accessible, deterministic environment. If the environment is inaccessible, however, then it may *appear* to be nondeterministic. This is particularly true if the environment is complex, making it hard to keep track of all the inaccessible aspects. Thus, it is often better to think of an environment as deterministic or nondeterministic *from the point of view of the agent.*

EPISODIC

- **Episodic** vs. **nonepisodic.**
  In an episodic environment, the agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

STATIC

- **Static** vs. **dynamic.**
  If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic**.

SEMIDYNAMIC

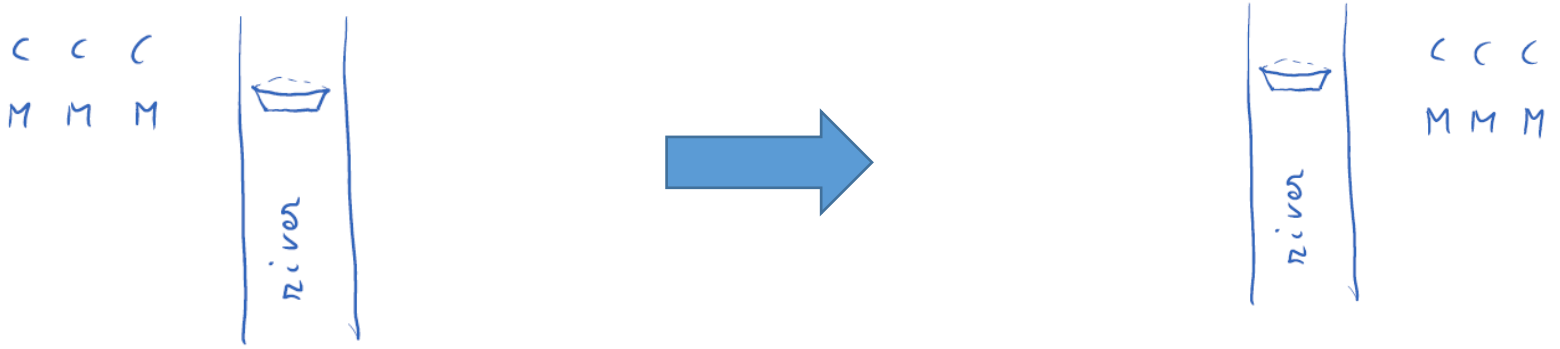
DISCRETE

- **Discrete** vs. **continuous.**
  If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Chess is discrete—there are a fixed number of possible moves on each turn. Taxi driving is continuous—the speed and location of the taxi and the other vehicles sweep through a range of continuous values.[10]

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

Figure 2.13 Examples of environments and their characteristics.

A little recreation: cannibals and missionaries



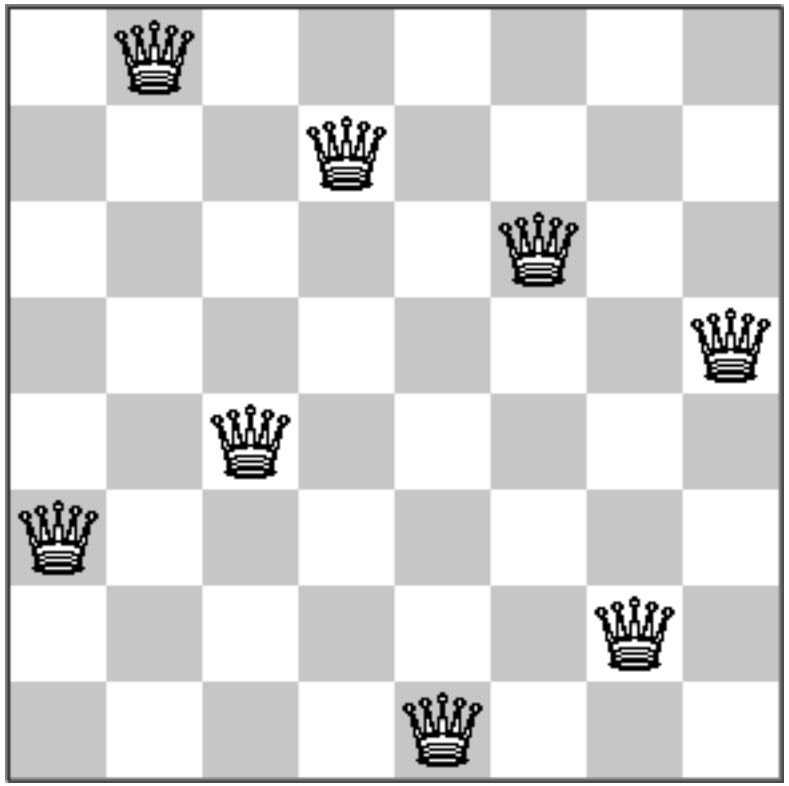
No more than 2 people on the boat
Number of cannibals <= number of missionaries on both sides of the river

Hint: list all «legal» states and then connect them with lines describing legal actions

Toy problems: the 8-puzzle and its companions

It is NP-complete
8-puzzle has 9!/2=181440 reachable states
15-puzzle has about 10^{13} states
24-puzzle has about 10^{25} states

Toy problem: 8-queens



Difficulty depends on the formulation:

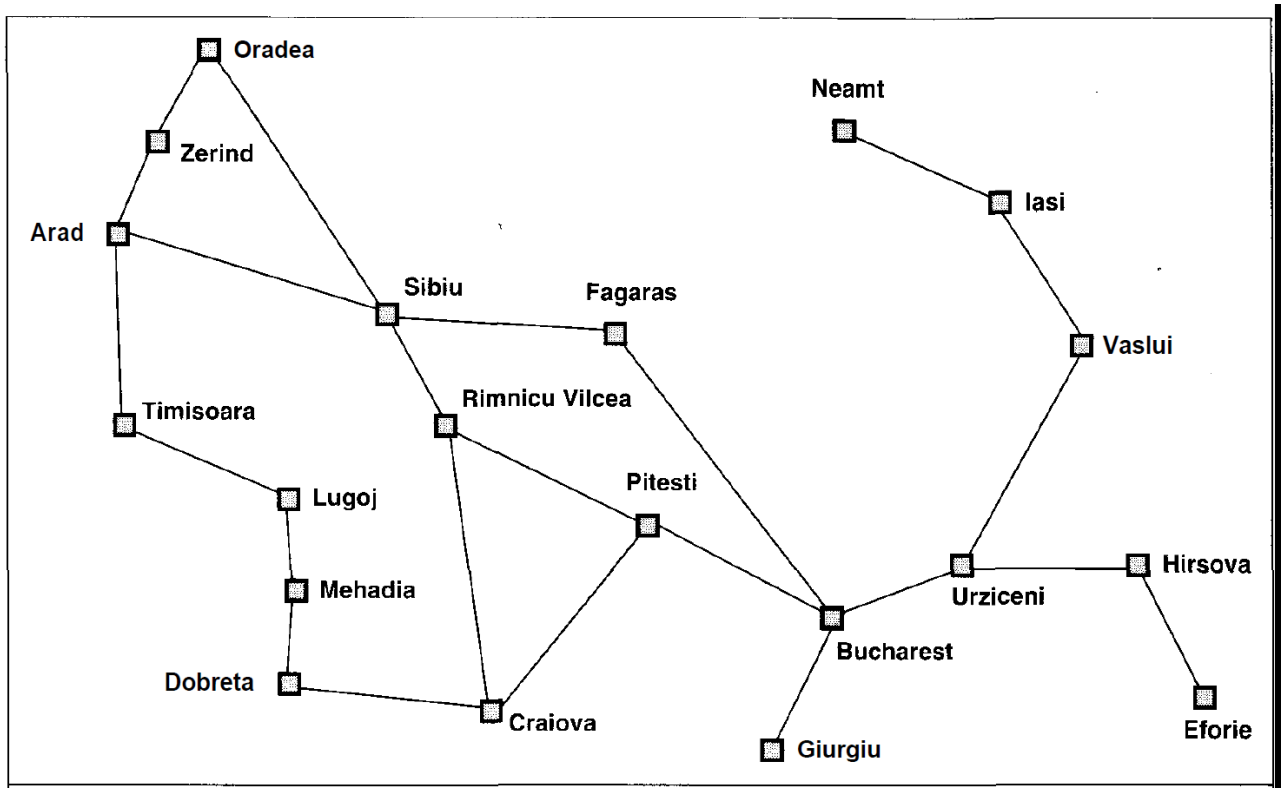
Empty board: 64x63x…x57 →10^{14} states;

8 queens already on the board all on different rows and columns: 8!=40320 states

100 queens problem: 100! States, still enormous

→ But even 1 million queens can be solved by using a simple (but dedicated) algorithm!

Real-world problems: route finding



Formulation of a problem

(we assume a static, fully accessible, discrete and deterministic environment)

- Initial state: I am in Timisoara
- Possible actions (action, successor): (go Arad, in Arad), (go Lugoj, in Lugoj)
- Goal test: in Bucarest?
- Path cost from x to y by the action a: c(x,a,y)>=0
- Optimal solution: the solution having the lowest cost

Searching for solutions

Check if the inital state is the goal. If not, we expand the current state, creating a search graph.

Definition of node. Five components:

- State: one element in state space (n)
- Parent-node: the node in the search tree that generated the present node
- Action: it was applied to the parent to generate the present node (n-1,a,n)
- Path-cost: total cost from the initial state to the node $g(n)= \sum_{i=1}^{n} c(i-1, a_i, i)$
- Depth: number of steps from initial state

Fringe: collection of nodes generated but not yet expanded

Parameters characterizing a problem:

- b: branching factor or maximum number of successors of any node
- d: depth of the shallowest goal node
- m: maximum lenght of any path in the state space

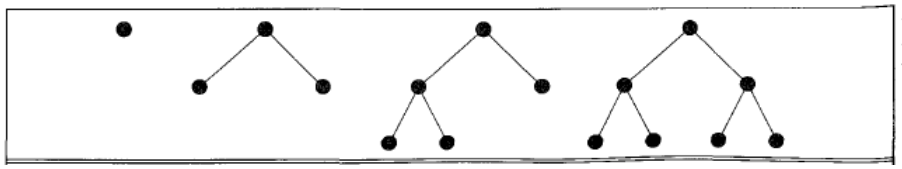
Qualifiers of an algorithm searching a solution:

- Completeness: does it find a solution if one solution exists?
- Optimality: does it find the lowest cost solution?
- Time needed: how long to find a solution?
- Memory needed: how much memory to perform the search?

Blind search

Breadth-first search

All nodes at depth p are expanded before going to depth p+1



- Complete (if b is finite)
- Not optimal, unless c(p+1)≥ c(p)
- All expanded nodes need to be kept in memory
- Time and memory $\sum_{p=1}^{d} b^p \approx b^{d+1}$

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Figure 3.12 Time and memory requirements for breadth-first search. The figures shown assume branching factor $b = 10$; 1000 nodes/second; 100 bytes/node.

Uniform-cost search

Instead of expanding the shallowest node, expands the node with the lowest path cost.

Identical to breadth-first if c=const.

To avoid infinite loops $c \geq \epsilon > 0$

Under that condition it is complete and optimal

If C* is the cost of the optimal solution,

time and memory needed scale as: $b^{C*/\epsilon}$

Depth-first search

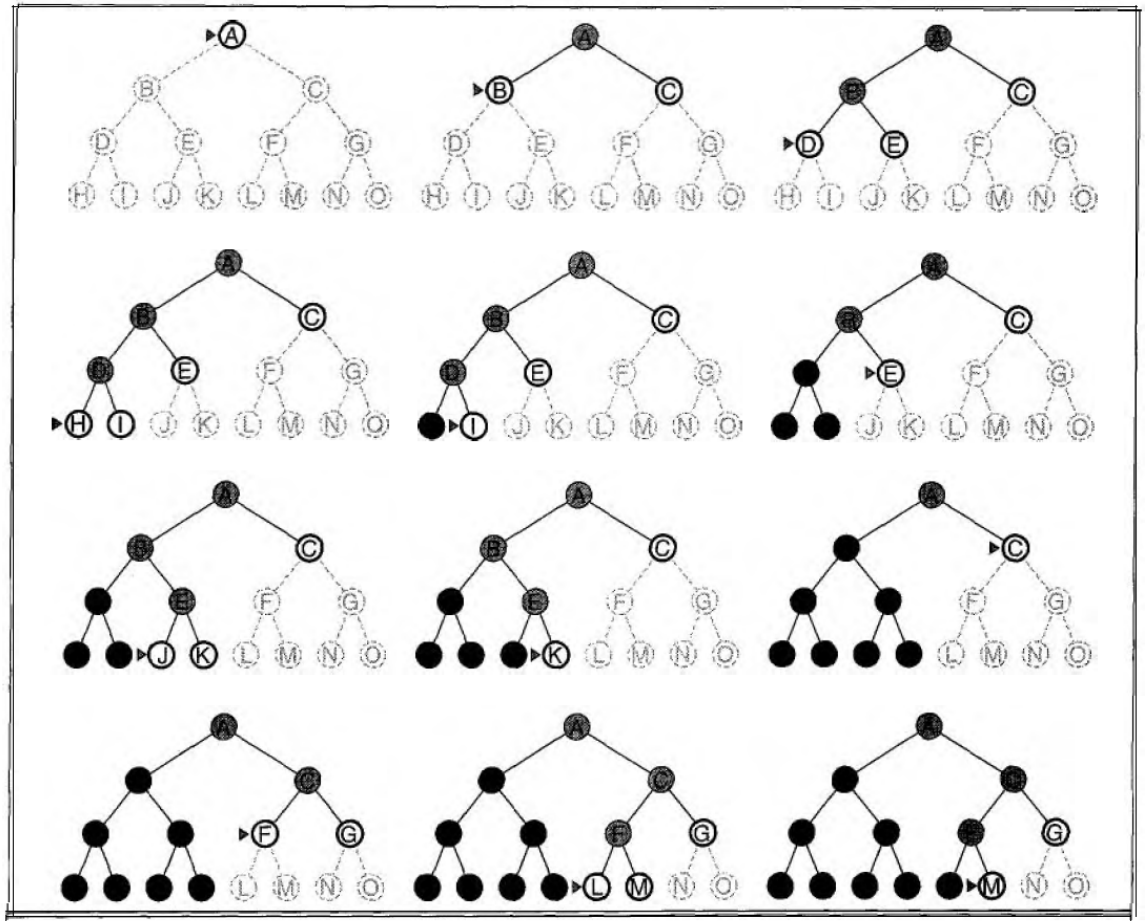


Figure 3.12 Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

All nodes not leading to a goal can be removed from memory.

Memory needed: ~ b m (if all successors are generated but not expanded)

Time needed: worst case ~ b^m where m (maximum depth) can be much larger than d (depth of the first solution)

It is not complete (if m → ∞)
It is not optimal

Depth-limited search

It is a depth-first search with a pre-determined depth limit l

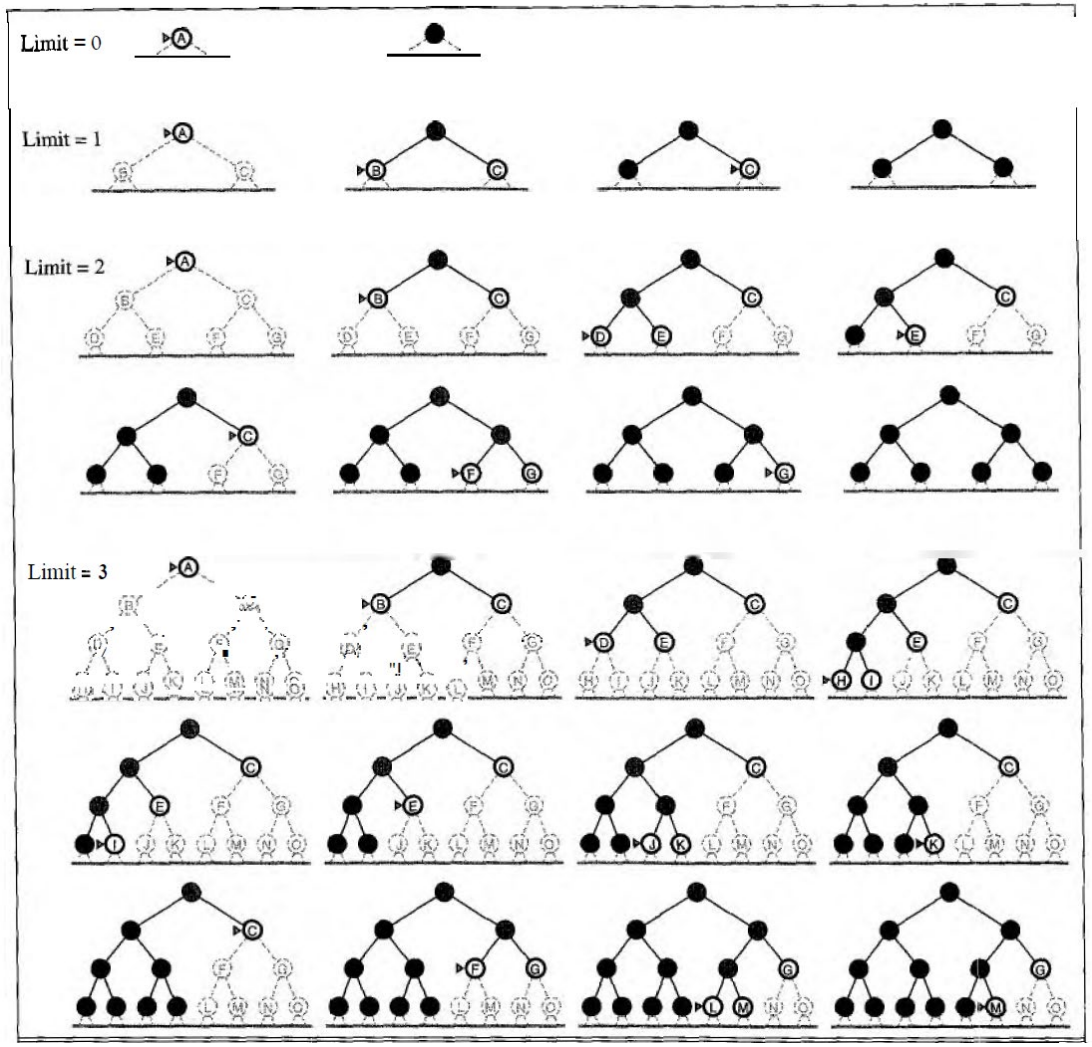
Memory needed $\sim b\ l$

Time needed $\sim b^l$

It is not complete (if $l < d$)

It is not optimal

Iterative deepening depth-first search



The limit l is gradually increased till a solution is found.

States are generated more than once, if b is more or less constant most of the nodes are at the bottom level.

Memory needed: $\sim b\ d$

Time needed: $\sim b^d$

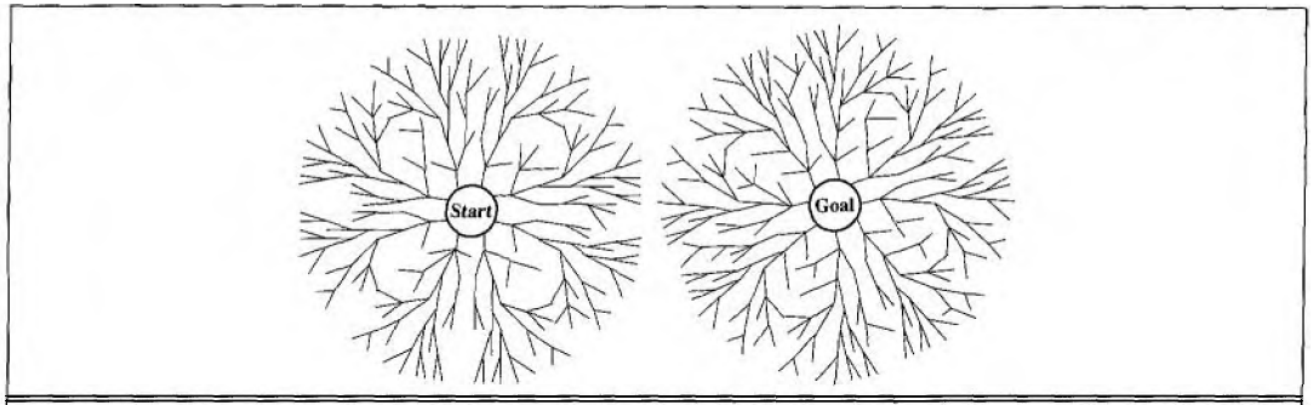
It is complete IF $b < \infty$

It is optimal IF $c(n+1) \geq c(n)$

Best strategy to explore large state spaces if the depth at which the solution is located is not known.

- *An alternative strategy of increasing path-cost limits can be formulated. It is not very efficent, but it constitutes the first step of the informed search A**

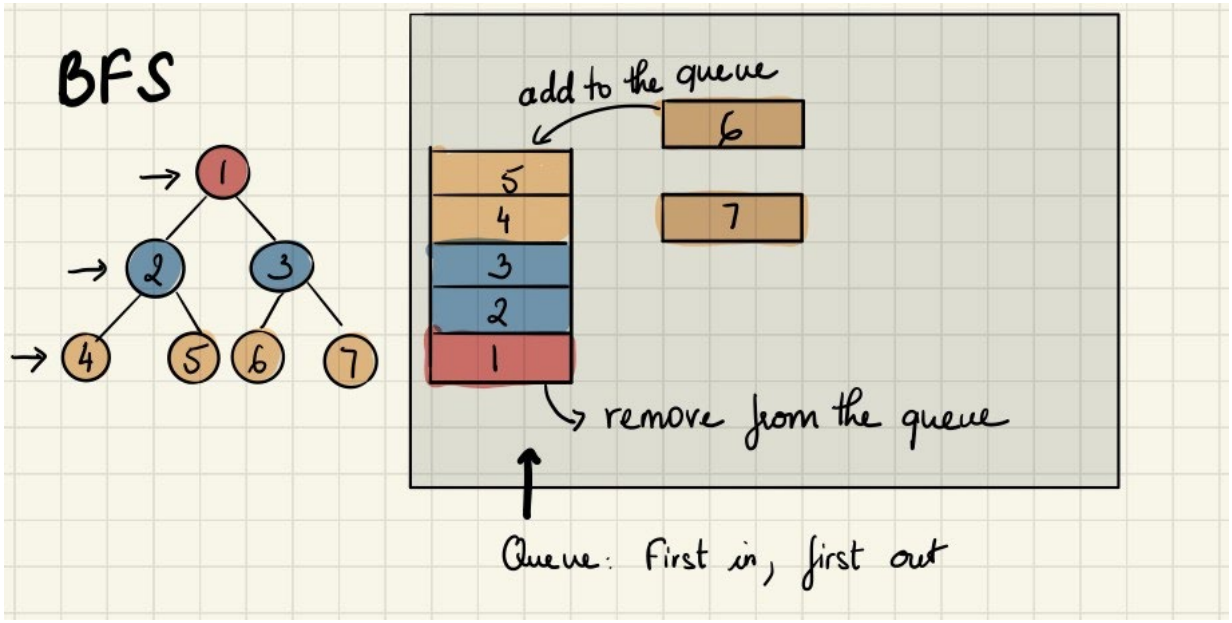
Bidirectional search



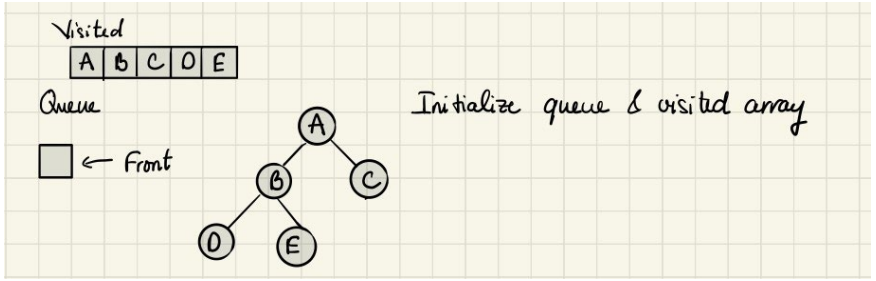
Idea: $b^{d/2}+b^{d/2} << b^{d}$

Not always possible: the goal need to be known

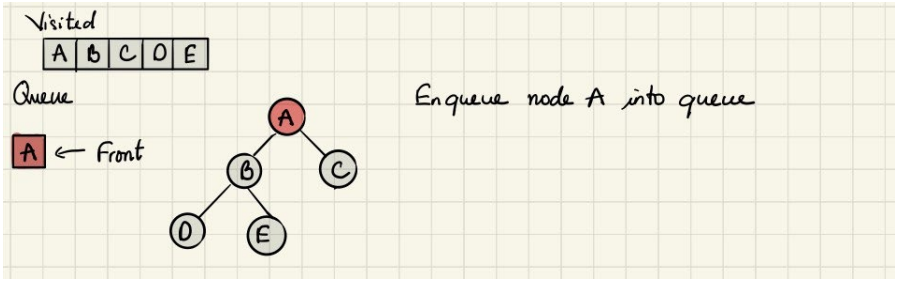
Breadth first

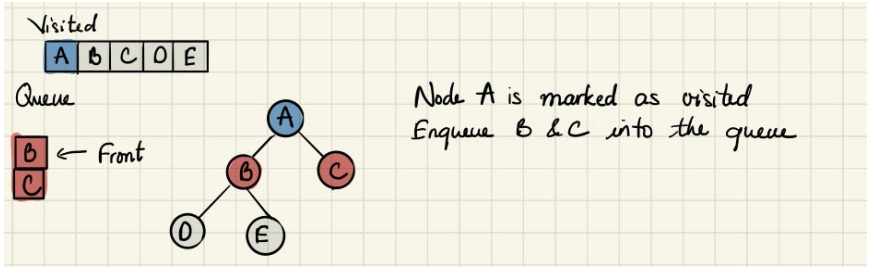


We want to use a data structure that, when queried, gives us the oldest element, based on the order they were inserted. A queue is what we need in this case since it is first-in-first-out(FIFO)



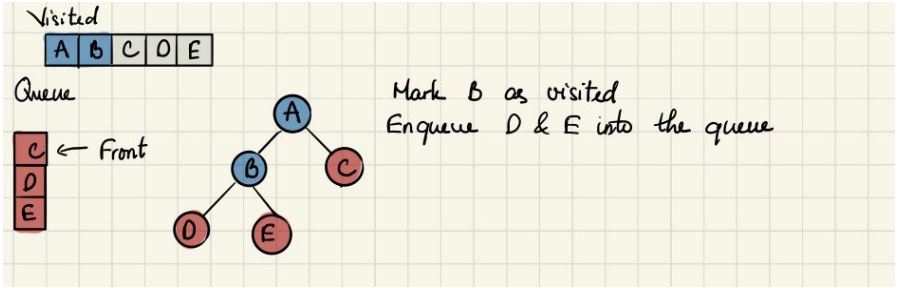

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : [],
  'D' : [],
  'E' : []
}
```

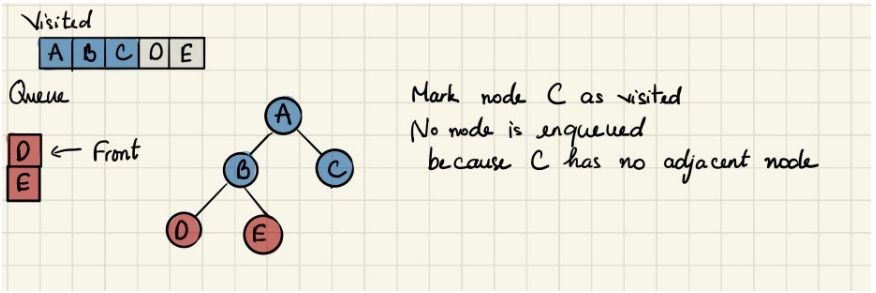




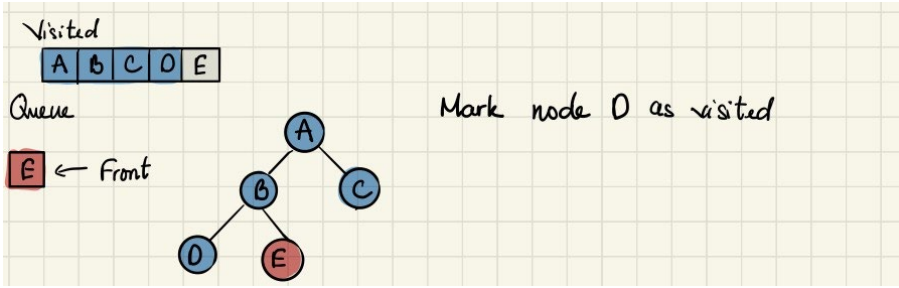
visited = []
List to keeps track of visited nodes.
queue is a list that is used to keep track of nodes currently in the queue.

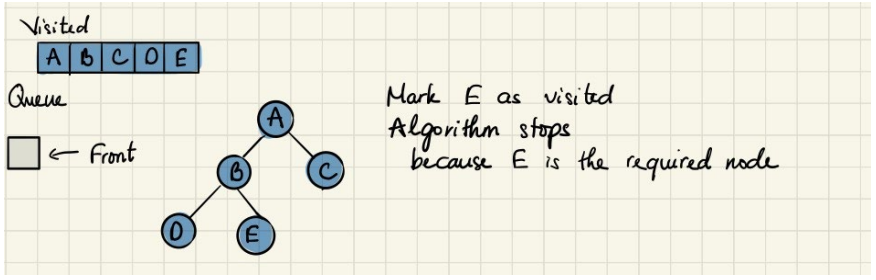
queue = [] #Initialize a queue






```
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    s = queue.pop(0)
    print (s, end = " ")

    for neighbor in graph[s]:
      if neighbor not in visited:
        visited.append(neighbor)
        queue.append(neighbor)
```





pop() is an inbuilt function in Python that removes and returns last value from the list or the given index value

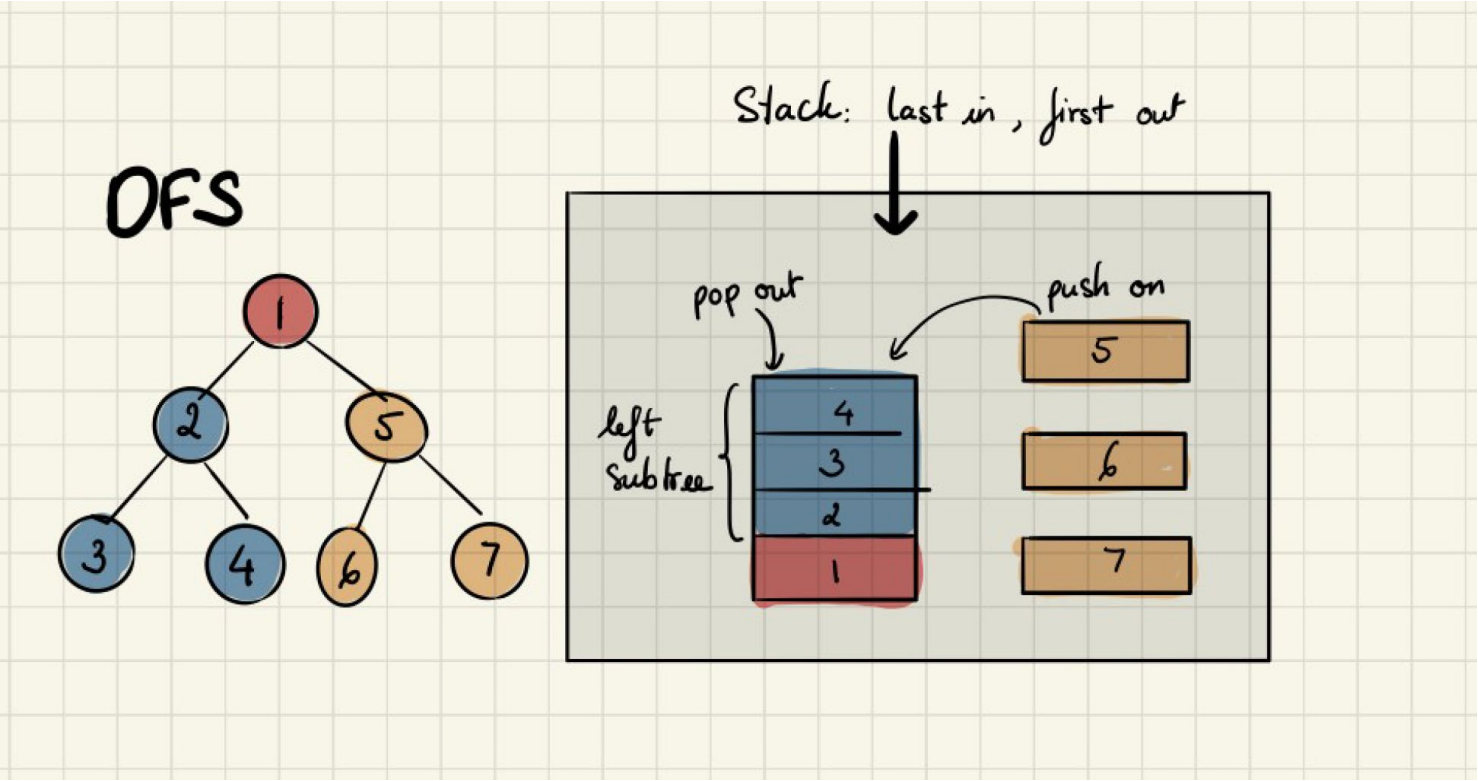
The purpose of “Visited” is to avoid loops, i.e. not to visit twice the same city if it can be reached in more than one way.

pop

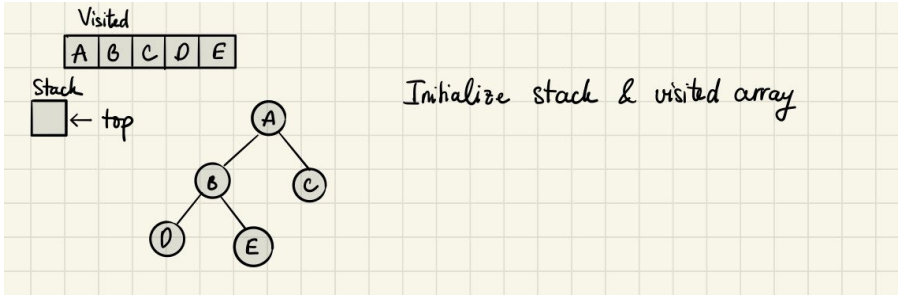
```
list1 = [ 1, 2, 3, 4, 5, 6 ]

# Pops and removes the last
# element from the list
print(list1.pop(), list1)

# Pops and removes the 0th index
# element from the list
print(list1.pop(0), list1)
```

```
6 [1, 2, 3, 4, 5]
1 [2, 3, 4, 5]
```

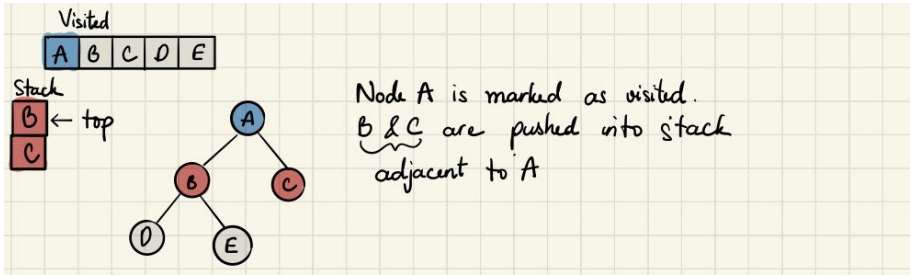
Depth first



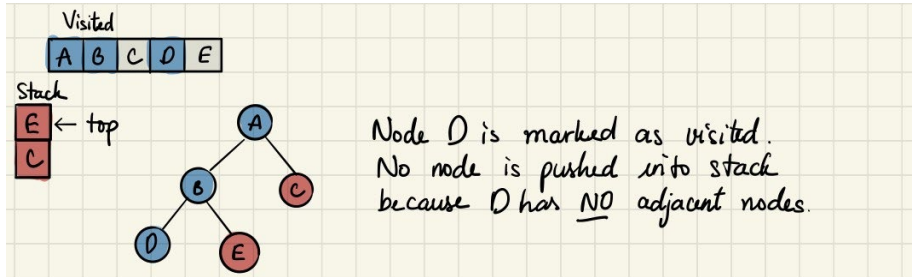
It requires last-in first-out approach, which can be implemented by the “stack”



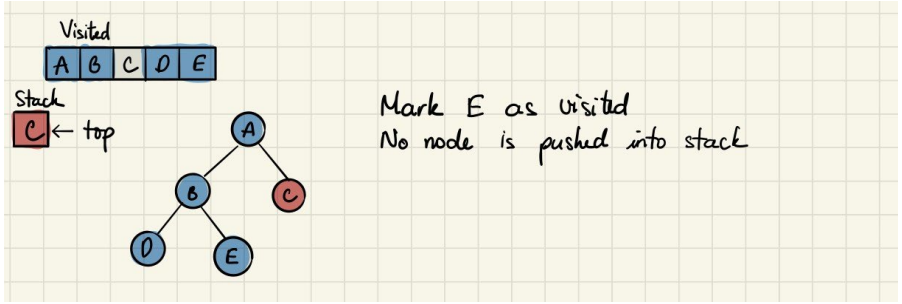
Initialize stack & visited array



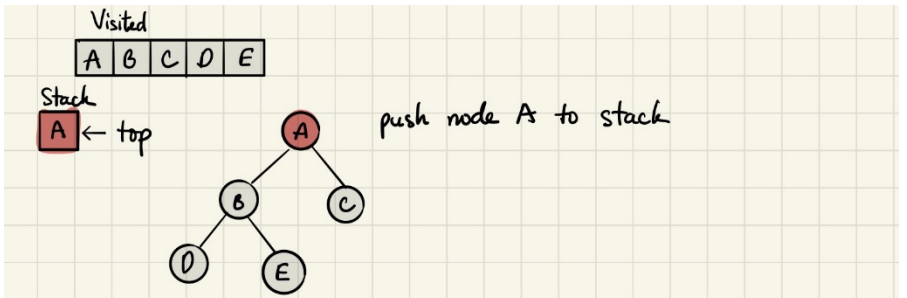
Node A is marked as visited.
B & C are pushed into stack
adjacent to A



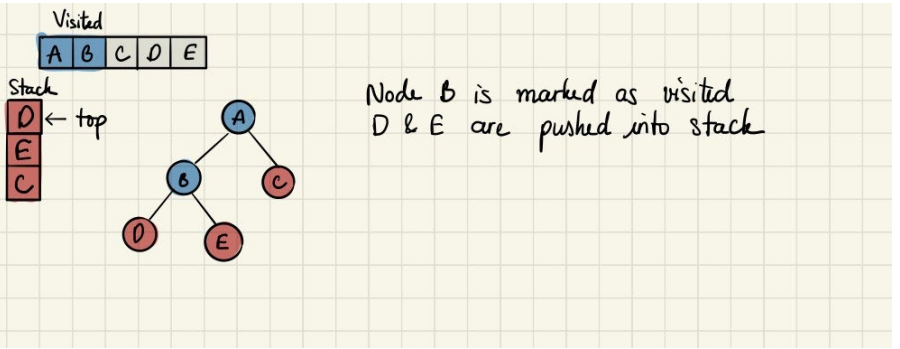
Node D is marked as visited.
No node is pushed into stack
because D has NO adjacent nodes.



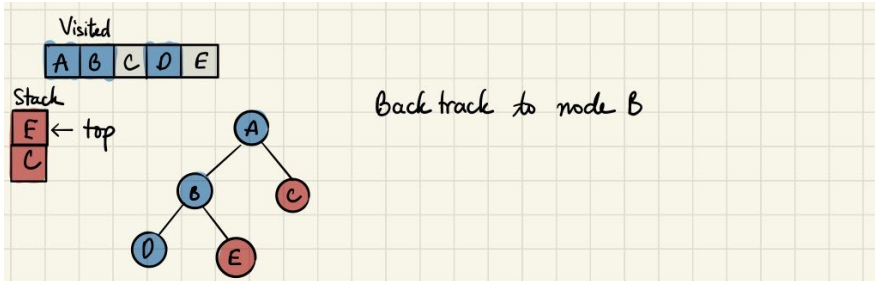
Mark E as visited
No node is pushed into stack



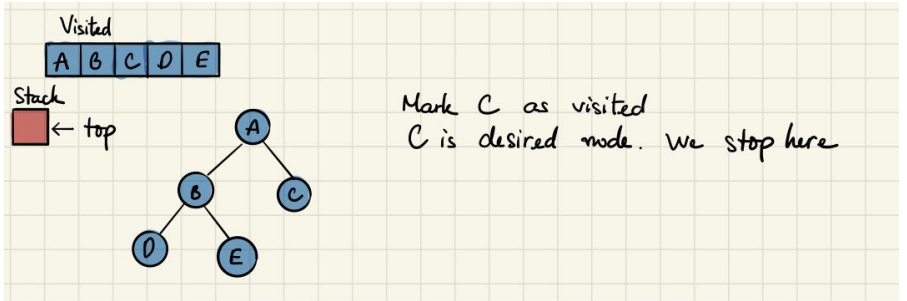
push node A to stack



Node B is marked as visited
D & E are pushed into stack



Back track to node B



Mark C as visited
C is desired node. We stop here

Informed (heuristic) search strategies

Heuristic function

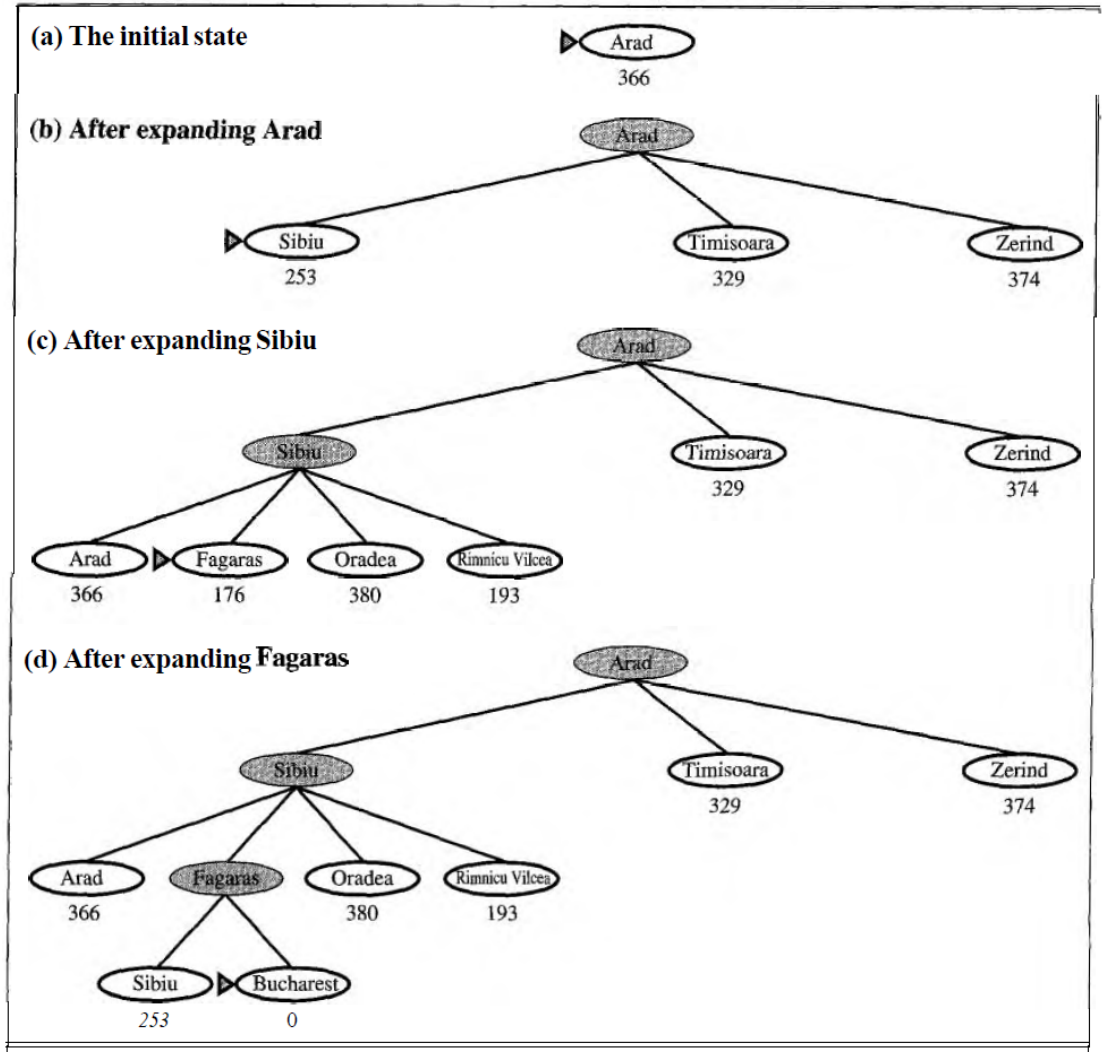
h(n): estimated cost of the cheapest path from node n to the goal

h(goal) = 0

Obviously IF we knew exactly the cost we would have solved the problem!

Greedy best-first search

We expand first nodes having the smallest h(n) down till the end



Similar to depth-first search:
It is not complete (infinite path)
It is not optimal

A* search: minimizing the total estimated cost

f(n) = g(n) + h(n)

g(n) is the REAL cost from start to n

h(n) is the estimated cost from n to goal

It is complete IF $c \geq \epsilon$ and IF b is finite

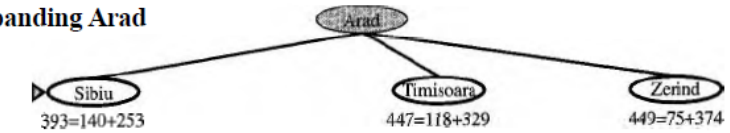
It is optimal if h is admissible: h(n) $\leq$ c(n,a,goal)

i.e. it DOES NOT overstimate the cost

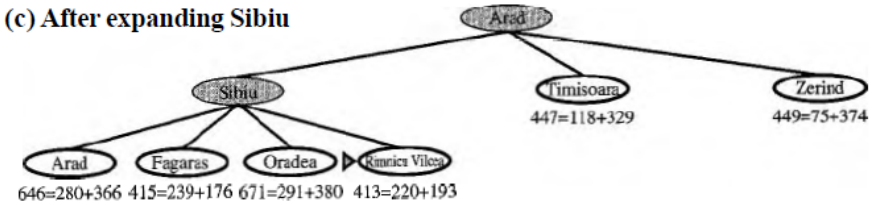
(a) The initial state

366=0+366

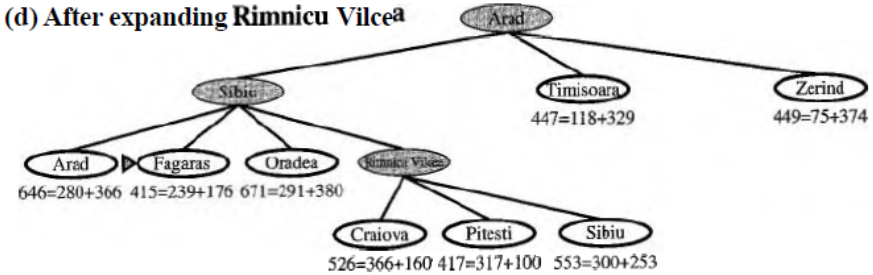
(b) After expanding Arad



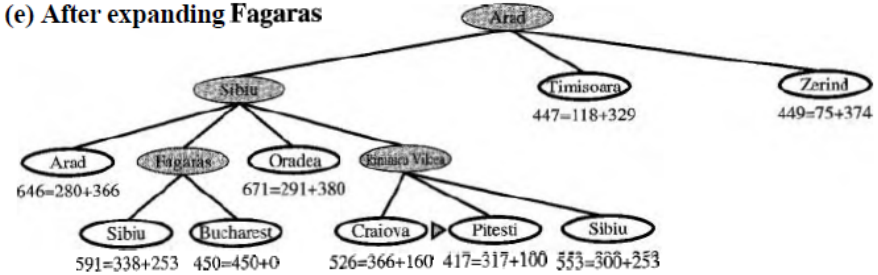
(c) After expanding Sibiu



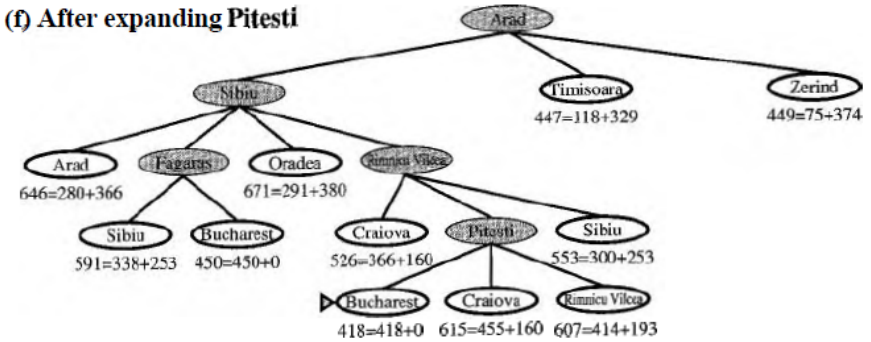
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



If h is admissible then A* is complete
(assuming $c \geq \varepsilon > 0$ and b finite)

The cost increases at least by ε at each step. Since b is finite the total cost will eventually exceed the cost of the solution if a solution exists.

A* is optimal if h is admissible

G_2 is a NOT optimal goal on the fringe. $h(G_2) = 0$

$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$ which is the cost of the optimal solution

Instead, let n be a node on the fringe along the optimal solution. Then:

$f(n) = g(n)+h(n) \leq g(n)+c(n,a,G^*) = C^*$

Therefore $f(n) \leq C^* < f(G_2)$ and therefore G_2 is not expanded

Consistent h

The heuristic function h is consistent if it satisfies a triangular inequality:

$h(n) \leq c(n,a,n') + h(n')$ for all n and n'

If h is consistent then it is admissible:

let a_i be a sequence of actions along an optimal solution. Then

$h(n) \leq c(n,a_1,n_1) + h(n_1) \leq c(n,a_1,n_1) + c(n_1,a_2,n_2)+h(n\ 2) \leq$ ...

$\leq c(n,a_1,n_1) + c(n_1,a_2,n_2)+\ldots+ c(n_{d-1},a_d,G^*)+h(G^*)=C(n,G^*)$

Therefore h is admissible.

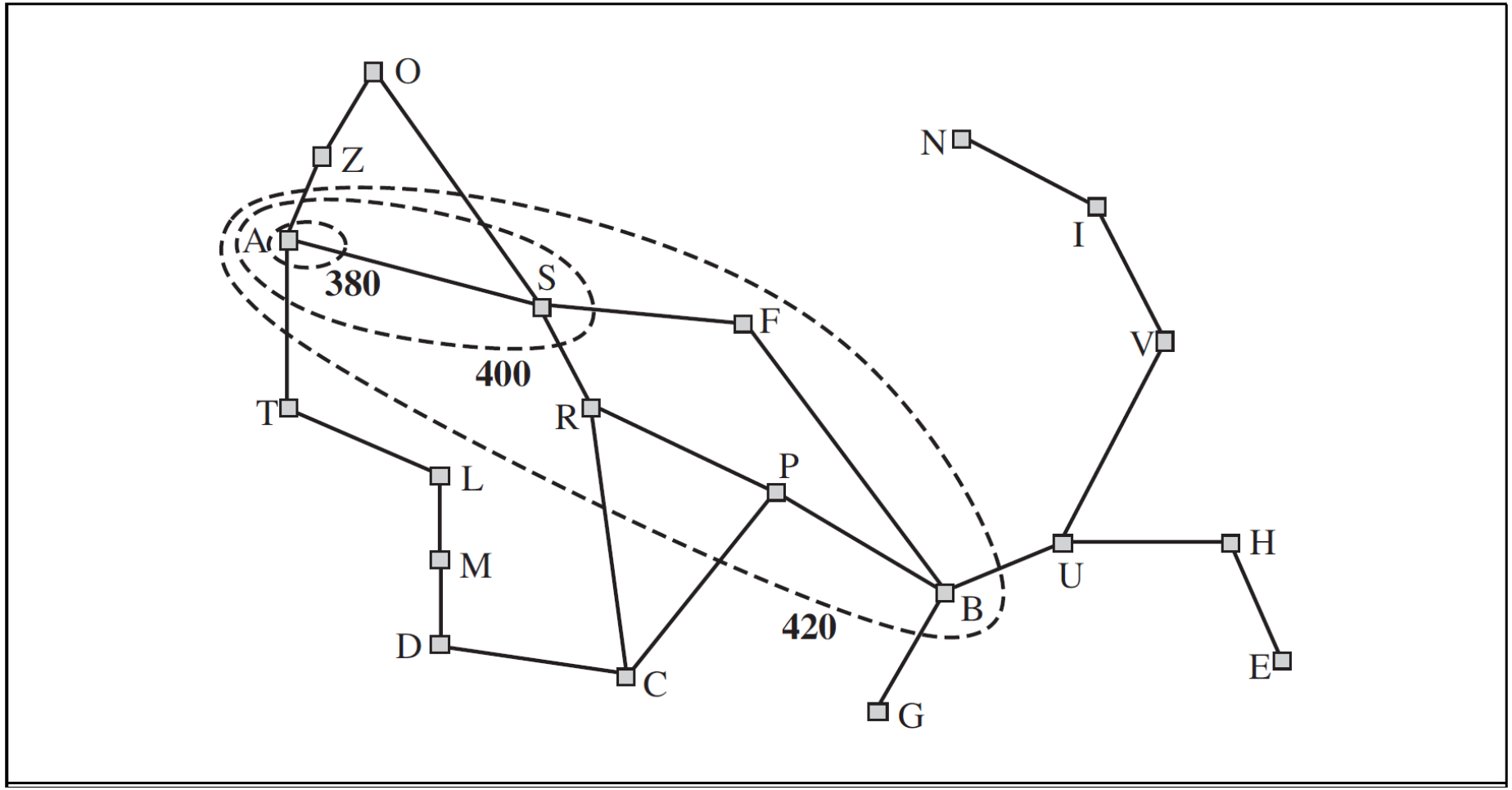
Again on optimality: if h is consistent then A* is optimal

Let a be an action conneting n to n', along whatever path. Then:

$f(n')=g(n')+h(n')=g(n)+c(n,a,n')+h(n') \geq g(n)+h(n)=f(n)$

Therefore f is a not-decreasing function (along whatever path). A* expands nodes having a not-decreasing cost and it expands ALL nodes having $f(n)<C^*$. Therefore the first goal it reaches has also the minimal cost.

In general the time needed still grows exponentially, unless h makes only logarithmic errors.

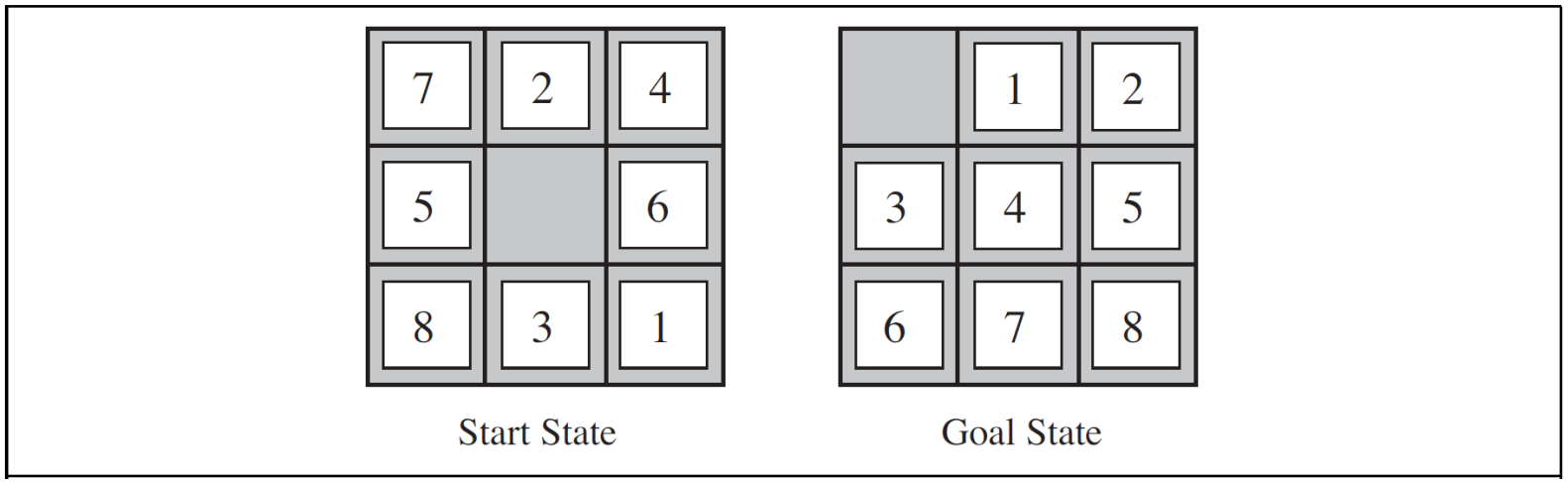

O
Z
A
380
S
F
400
T
R
L
P
M
D
C
420
B
G
N
I
V
U
H
E

Inventing admissible heuristic functions

The cost of an optimal solution to a «relaxed» problem is an admissible heuristic function for the original problem.

8-puzzle:

1. Tile can jump: h_1 counts the number of missplaced tiles
2. Tile can move one step also over already occupied squares:
   h_2 counts the number of steps (Manhattan distance)



$C^* = 26$
$h_1 = 8$
$h_2 = 18$

$h_2(n) \geq h_1(n)$
h_2 «dominates» h_1

Comparing Iterative Deepening search with A*

Nodes generated $N+1 = 1 + b^* + b^{*2} + \ldots + b^{*d}$

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Learning heuristic from experience

By solving many times a problem one can find correlations between certain characteristics of a state and the cost of the solution. E.g. a correlation between the number of misplaced tales and the cost.

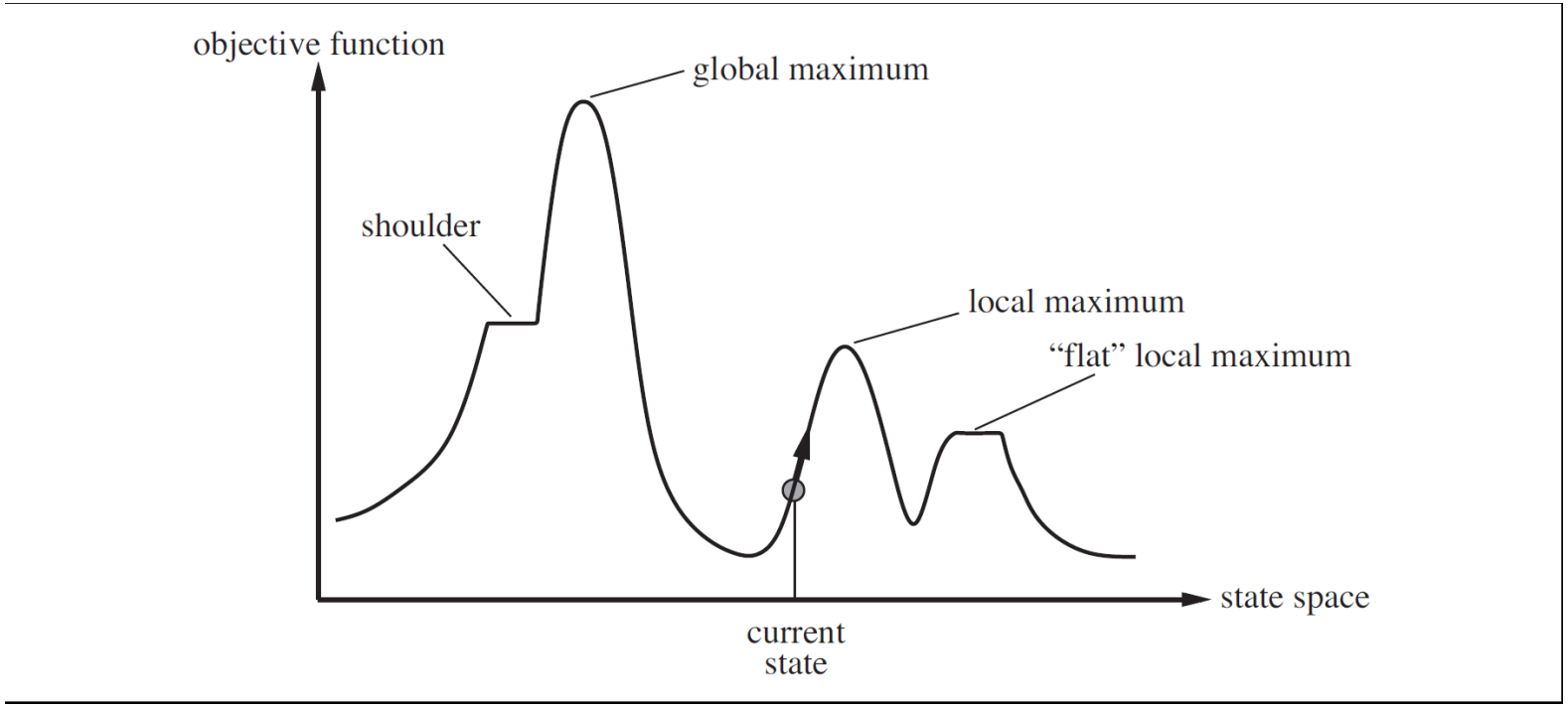
Learning strategies, e.g. neural networks, can be used to extract this type of information and produce heuristic functions.

Local search

In many problems the path to the solution is irrelevant, for instance in problems of optimization.

If path is not needed:

- Memory request are modest
- It is possible to explore large spaces



Ridges

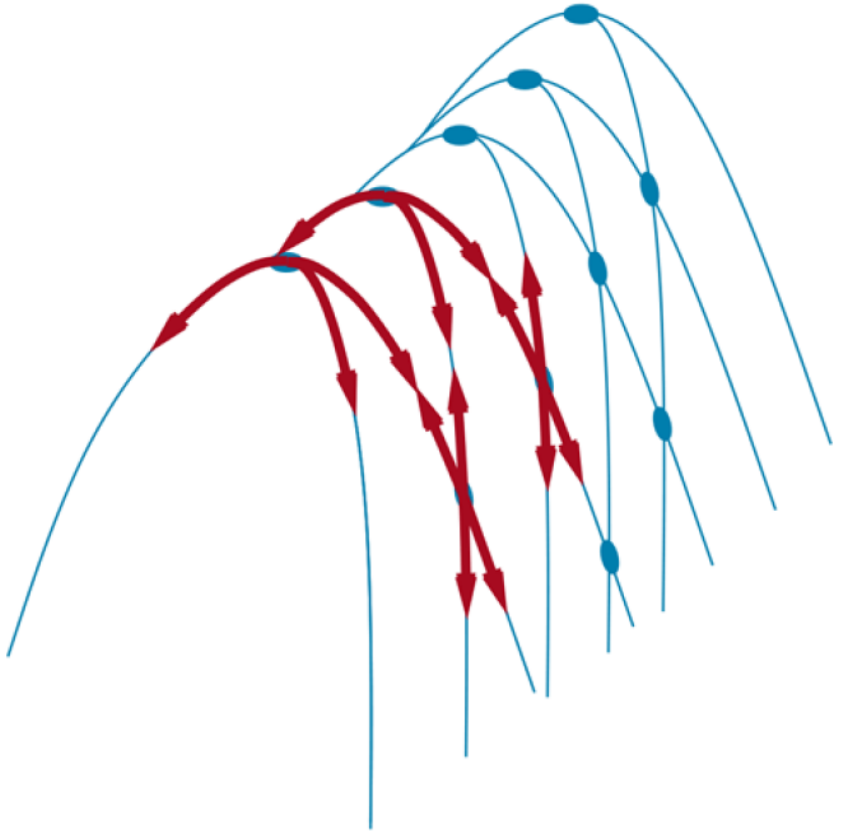
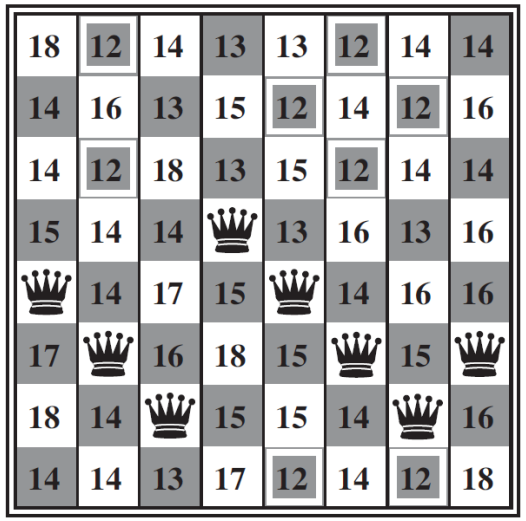
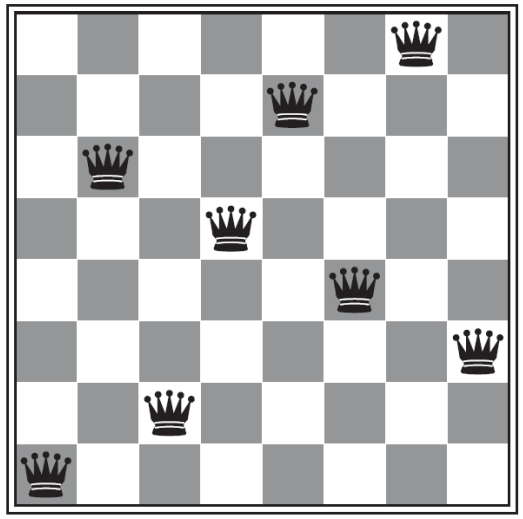


Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill. Topologies like this are common in low-dimensional state spaces, such as points in a two-dimensional plane. But in state spaces with hundreds or thousands of dimensions, this intuitive picture does not hold, and there are usually at least a few dimensions that make it possible to escape from ridges and plateaus.

A local minimum: greedy search would get stuck



(a)



(b)

Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Strategies for local search

- Random restart hill-climbing: restarts from random generated initial states
- Montecarlo techniques, simulated annealing: random walk and down-hill moves
- Local beam search: generates all successors of k replica, then selects the best k successors from the complete list
- Stochastic beam search: chooses randomly k successors from the complete list, with probability increasing with its value
- Genetic algorithms: combines two parents and tries to mimic natural selection

Genetic algorithms

Non attacking pairs of queens