TDDC17 LE2 HT2023 – Search I

Fredrik Heintz

Dept. of Computer Science
Linköping University

fredrik.heintz@liu.se
@FredrikHeintz

Outline:

- **Physical Symbol Systems**
- **Heuristic Search Hypothesis**
- **Search I: Uninformed Search Algorithms (Ch 3)**

LINKÖPING UNIVERSITY

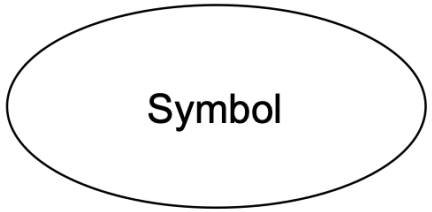
Physical Symbol System Hypothesis

Computer Science as Empirical Enquiry: Symbols and Search
Newell and Simon (1976)

Newell and Simon are trying to lay the foundational basis for the science of artificial intelligence.

What are the structural requirements for intelligence?

Can we define laws of qualitative structure for the systems being studied?



What is a symbol, that intelligence may use it, and intelligence, that it may use a symbol? (McCulloch)

Physical Symbol Systems

The adjective "physical" denotes two important aspects:
- Such systems clearly obey the laws of physics -- they are realizable by engineered systems made of engineered components.
- The use of the term "symbol" is not restricted to human symbol systems.

A physical symbol system consists of:
- a set of entities called symbols which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure).
- At any instant of time the system will contain a collection of symbol structures.
- The system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction, and destruction.

A physical-symbol system is a machine that produces through time an evolving collection of symbol structures and exists in a world of objects wider than just those symbol structures themselves.

Designation & Interpretation

There are two concepts central to these structures of expressions, symbols and objects:

Designation - An expression designates an object if, given the expression, the system can either effect the object itself or behave in ways depending on the object. [Causal relationship between expression and object]

Interpretation - The system can interpret an expression if the expression designates a process and if, given the expression, the system can carry out the process. [Expressions of a certain type can evoke processes]

Some additional requirements in the paper

Physical-Symbol System Hypothesis

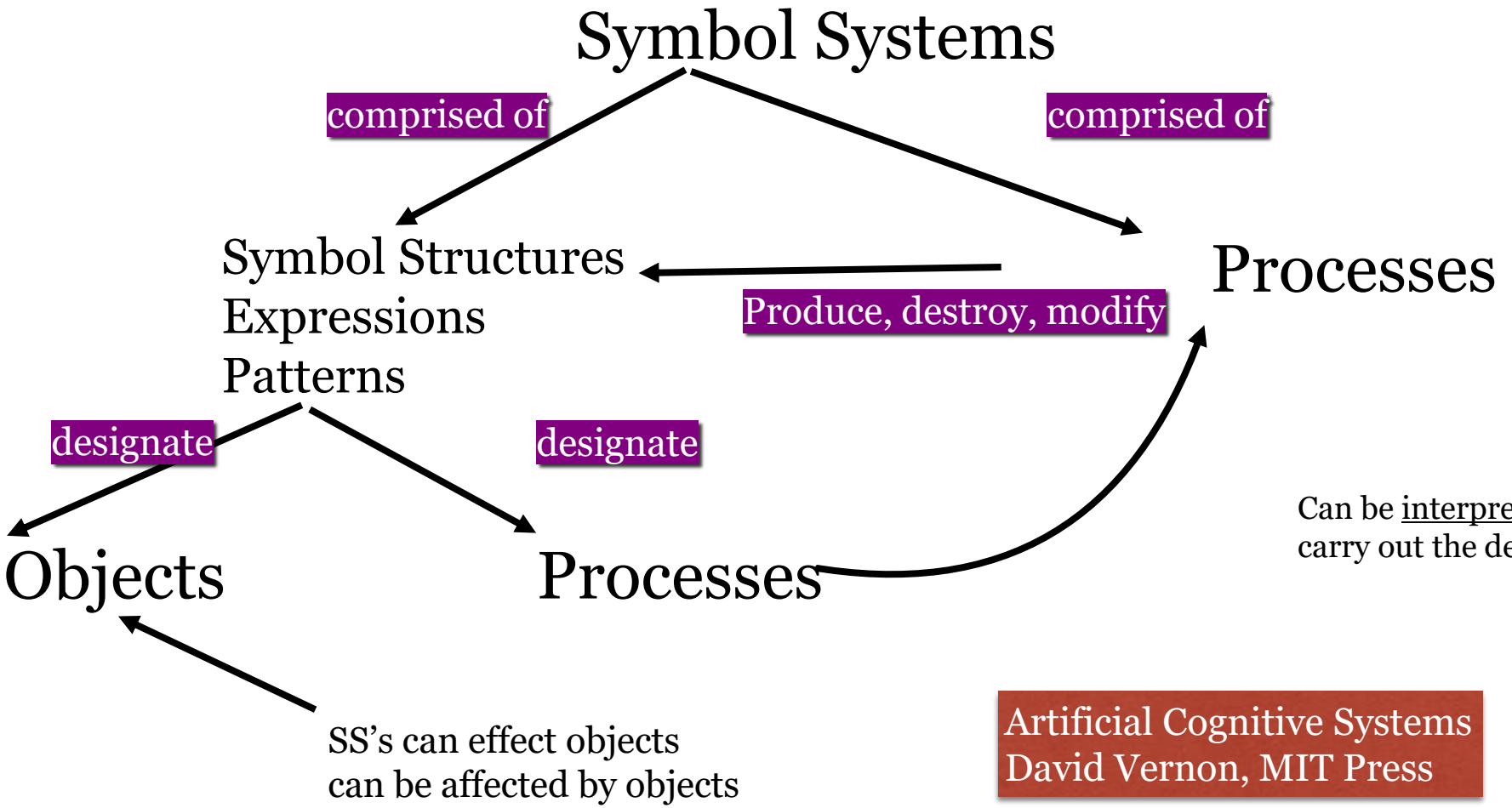
The Physical-Symbol System Hypothesis - A physical-symbol system has the necessary and sufficient means for general intelligent action.

necessary - any system exhibiting intelligence will prove upon analysis to be a physical symbol system.

sufficient - any physical-symbol system of sufficient size can be organized further to exhibit general intelligence.

Controversial!

PSS Hypothesis: Graphical Summary



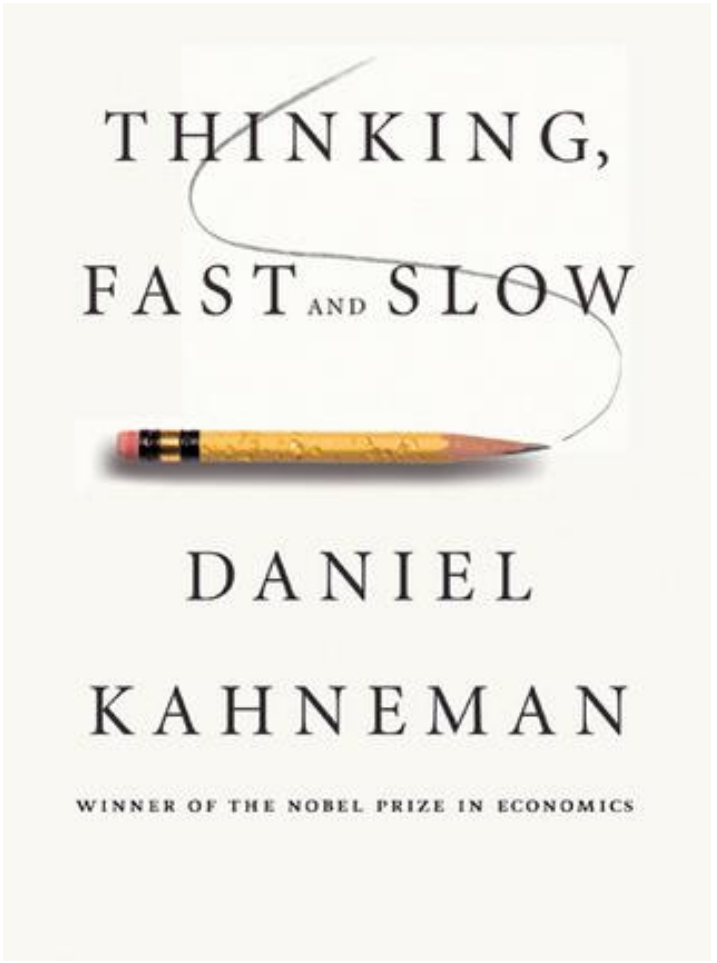
Can be interpreted:
carry out the designated processes

Artificial Cognitive Systems
David Vernon, MIT Press

Human and Computational Thinking

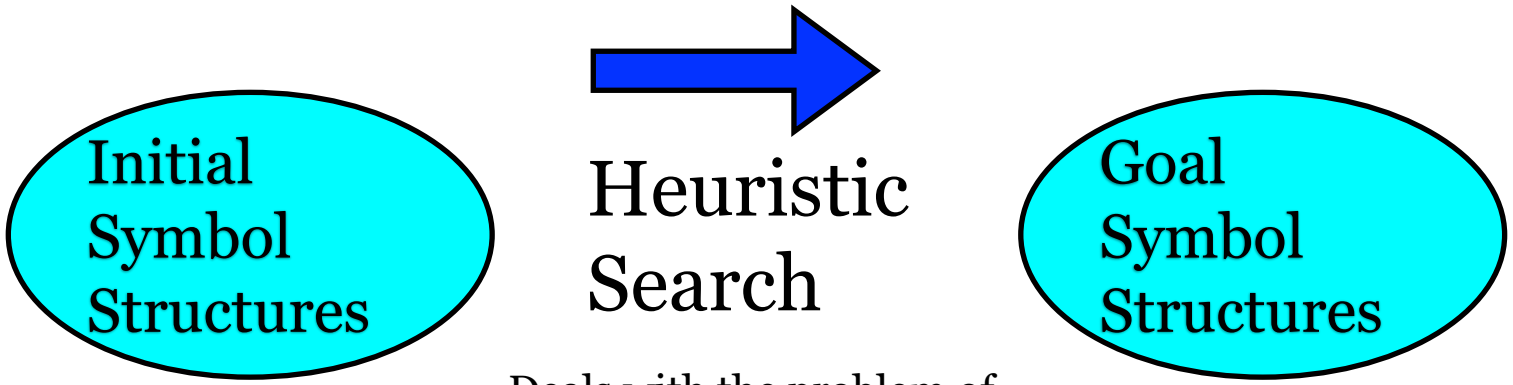
Figure 1: A Comparison of System 1 and System 2 Thinking

System 1 "Fast"	System 2 "Slow"
DEFINING CHARACTERISTICS Unconscious Effortless Automatic	DEFINING CHARACTERISTICS Deliberate and conscious Effortful Controlled mental process
WITHOUT self-awareness or control	WITH self-awareness or control
"What you see is all there is."	Logical and skeptical
ROLE Assesses the situation Delivers updates	ROLE Seeks new/missing information Makes decisions



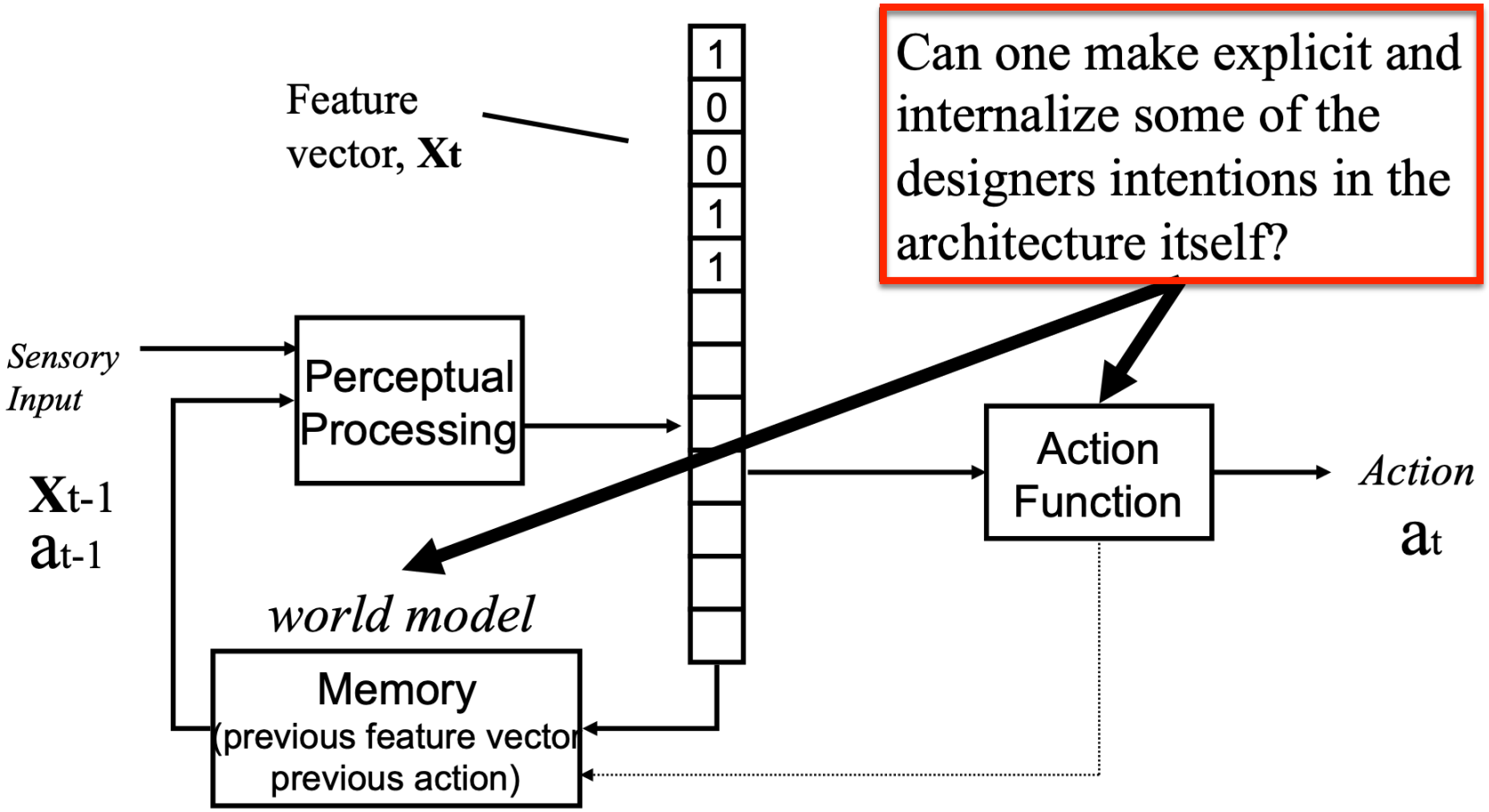
Heuristic Search Hypothesis

Heuristic Search Hypothesis - The solutions to problems are represented as symbol structures. A physical-symbol system exercises its intelligence in problem solving by search -- that is, by progressively modifying symbol structures until it produces a solution structure.

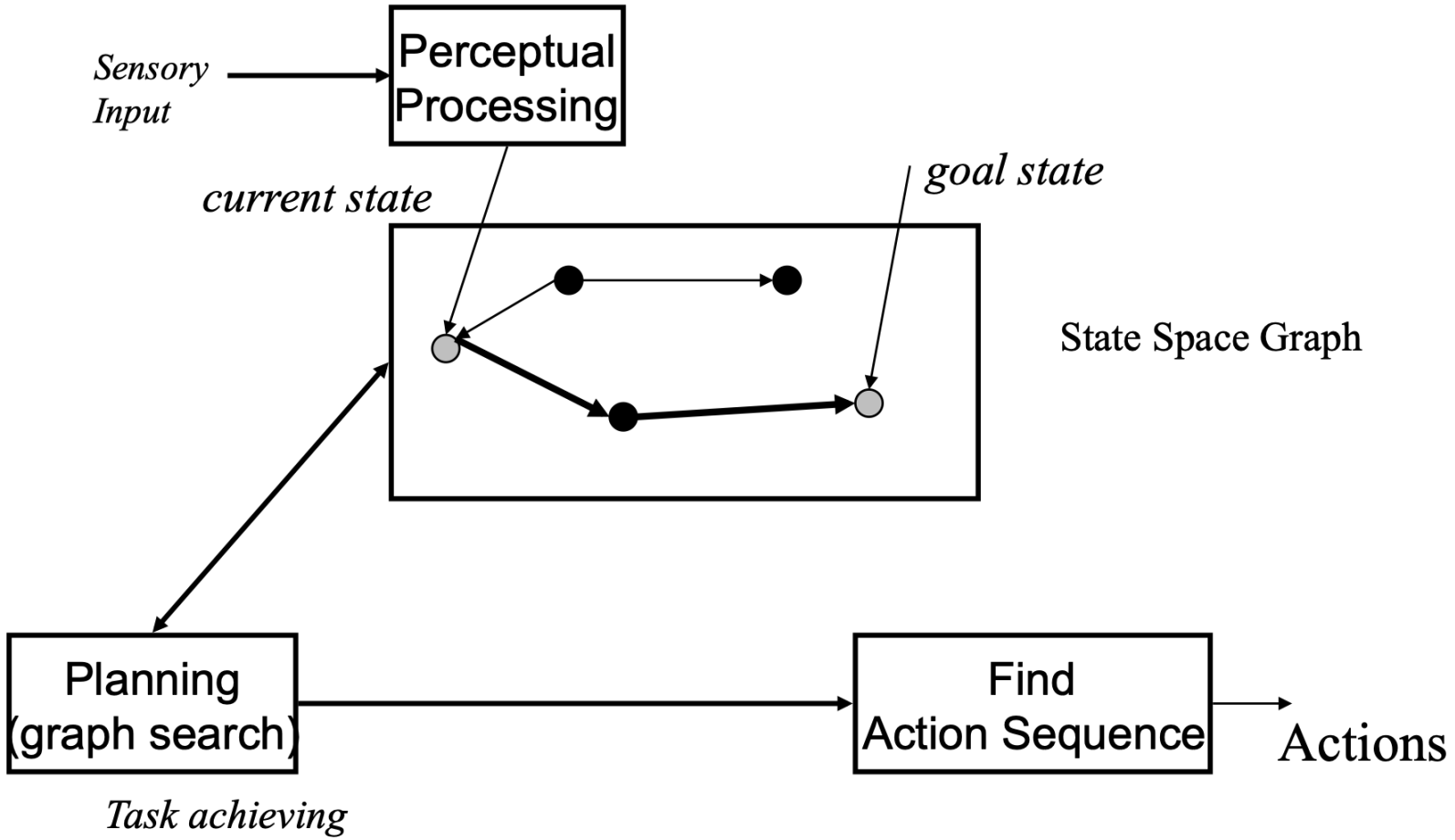


Deals with the problem of
Limited computational resources
The idea of “satisficing”

Recall: State Machine Agent

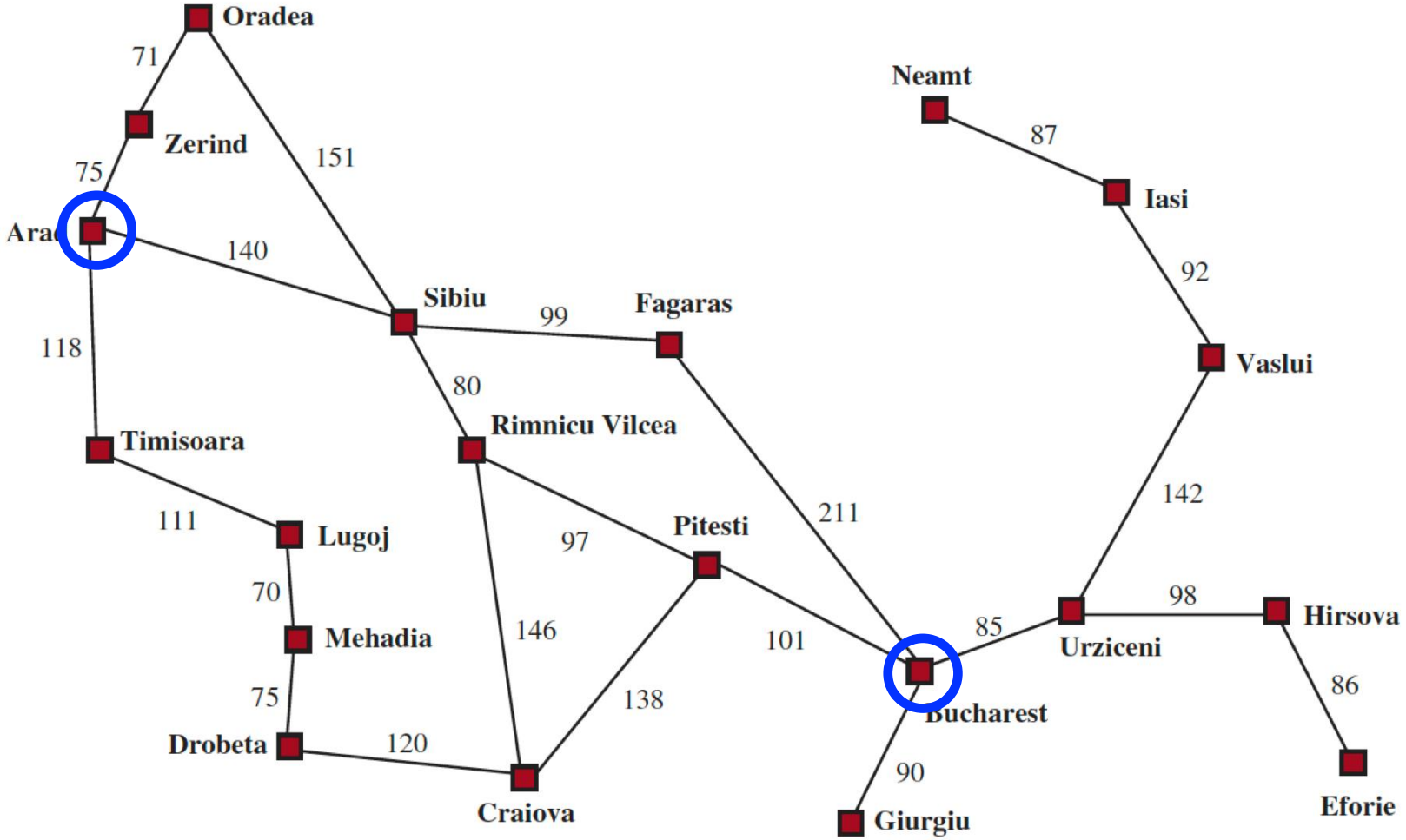


Problem Solving Agent: Version I

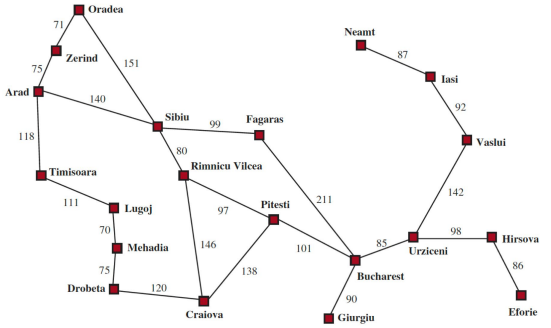


Algorithmic, Knowledge-Based and Learning-Based AI

Romania Route Finding Problem



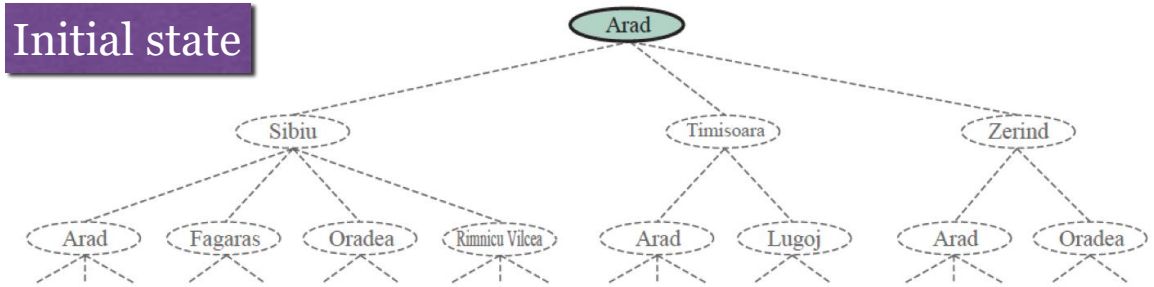
Problem Formulation



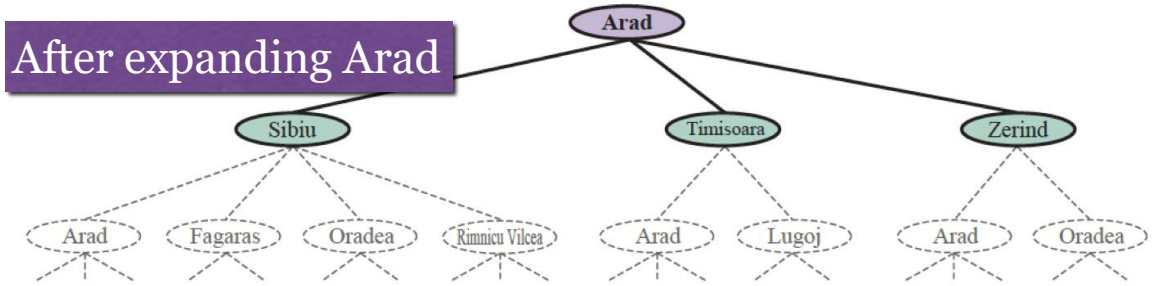
- State Space - A set of possible states that the environment can be in
  - Cities in the Romania map.
- Initial State - The state the agent starts in
  - Arad
- **ACTIONS**(*State*) - A description of what actions are applicable in each state.
  - **ACTIONS**(Arad)= {ToSibiu, ToTimisoara, ToZerind}
- **RESULT**(*State, Action*) - A description of what each action does (Transition Model)
  - **RESULT**(Arad, ToZerind)= Zerind
- Goal Test - Tests whether a given state is a goal
  - Often a set of states: { Bucharest }
- An Action Cost Function - denoted **ACTION-COST**(*s,a,s'*) when programming, or *c(s,a,s')* when doing math.
  - Gives the numeric cost of doing *a* in *s* to reach state *s'*.
  - Cost functions should reflect the agents performance measure: distance, time taken, etc.
- Solution - A path from the start state to the goal state

3 Partial Search Trees: Route Finding

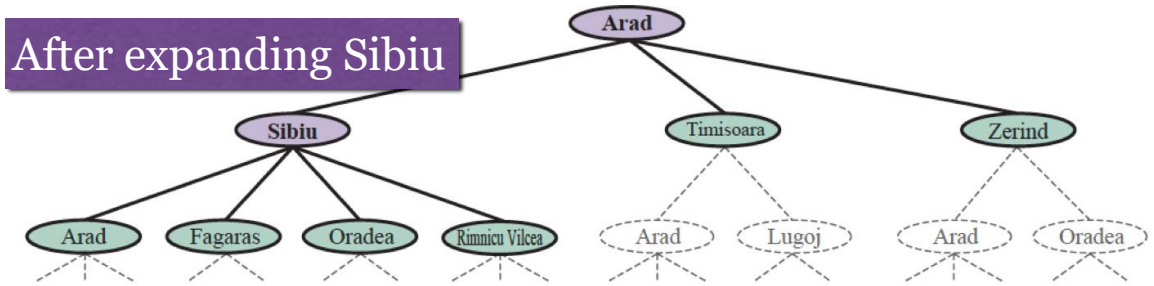
Initial state



After expanding Arad



After expanding Sibiu



Expanded

Generated

Not Expanded

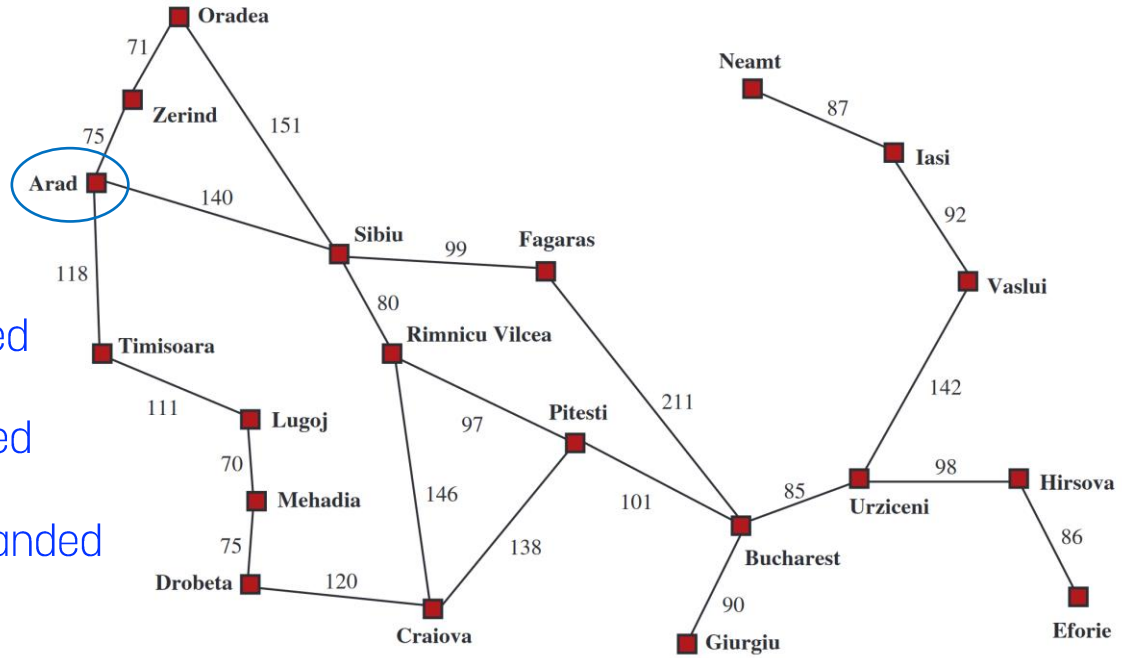
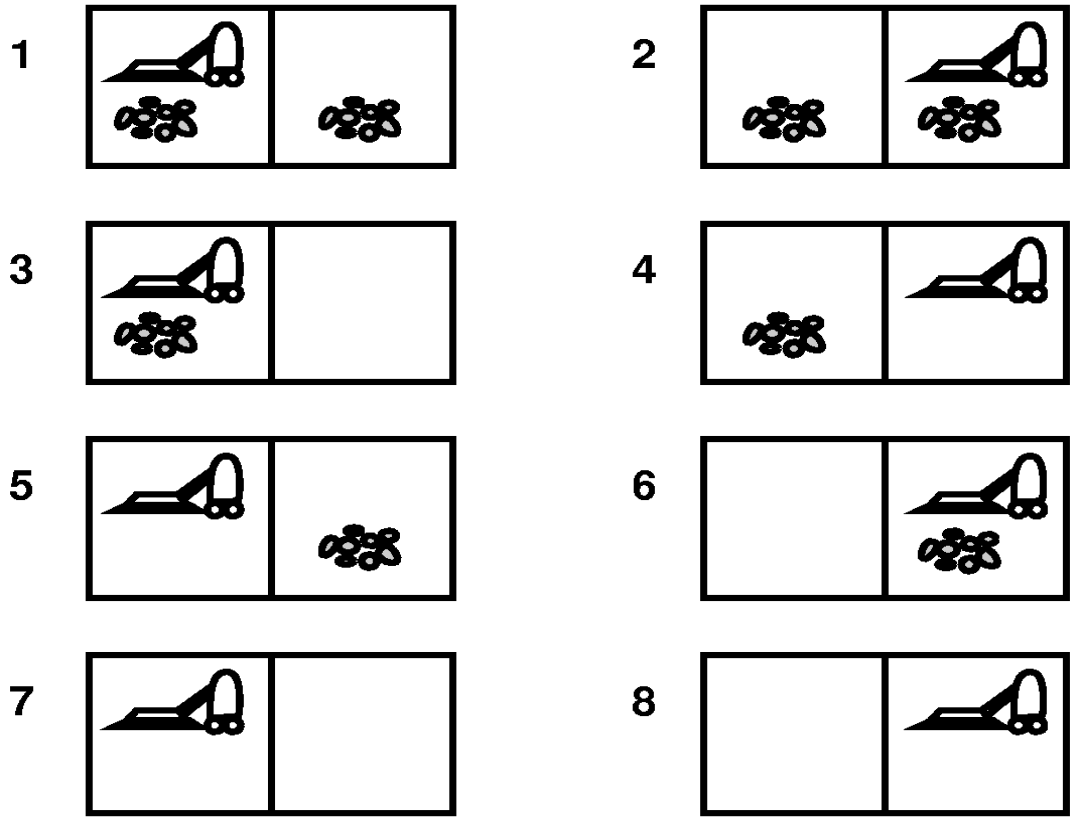


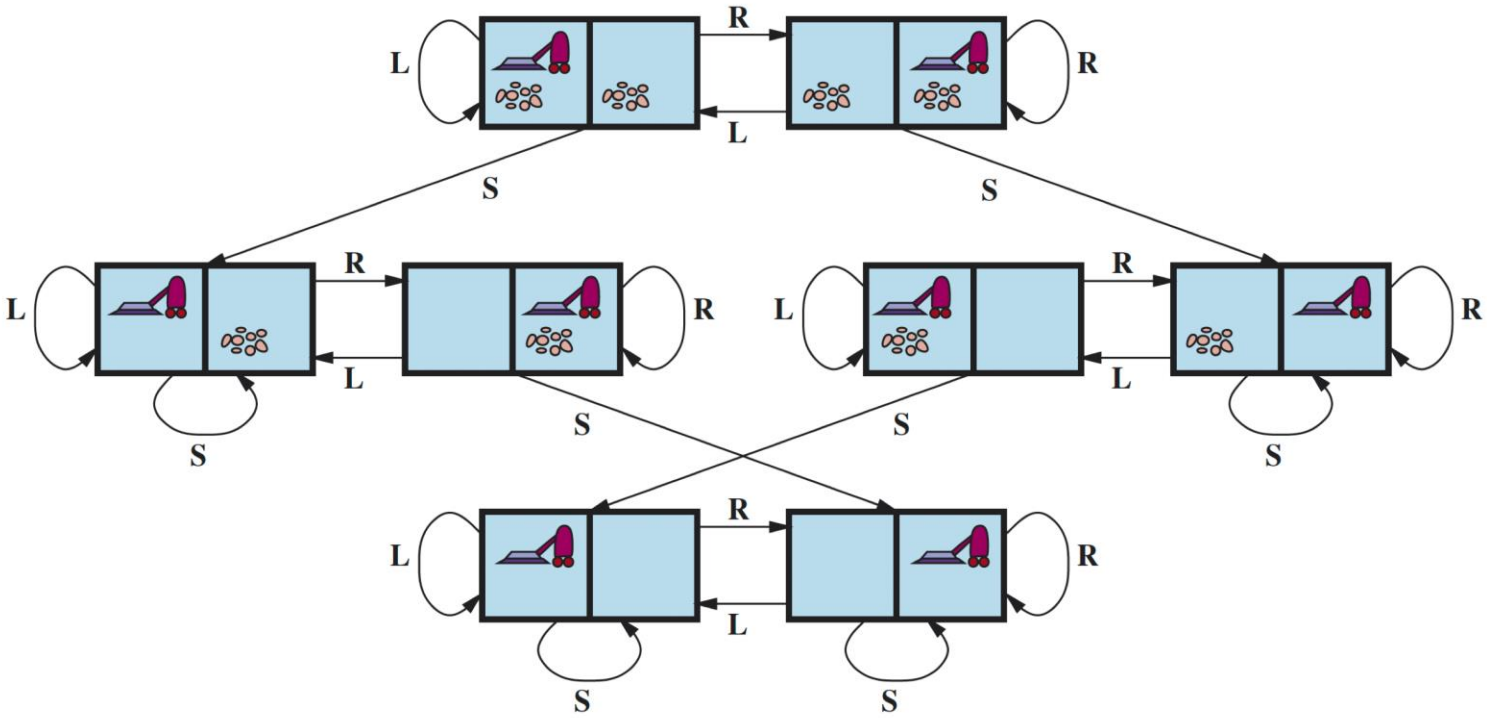
Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Problem Formulation: Vacuum cleaning world

- State Space:
  2 positions, dirt or no dirt
  -> 8 world states
- Actions:
  Left (L), Right (R), or Suck (S)
- Transition model: next slide
- Initial State: Choose.
- Goal States:
  States with no dirt in the rooms
- Action costs:
  one unit per action



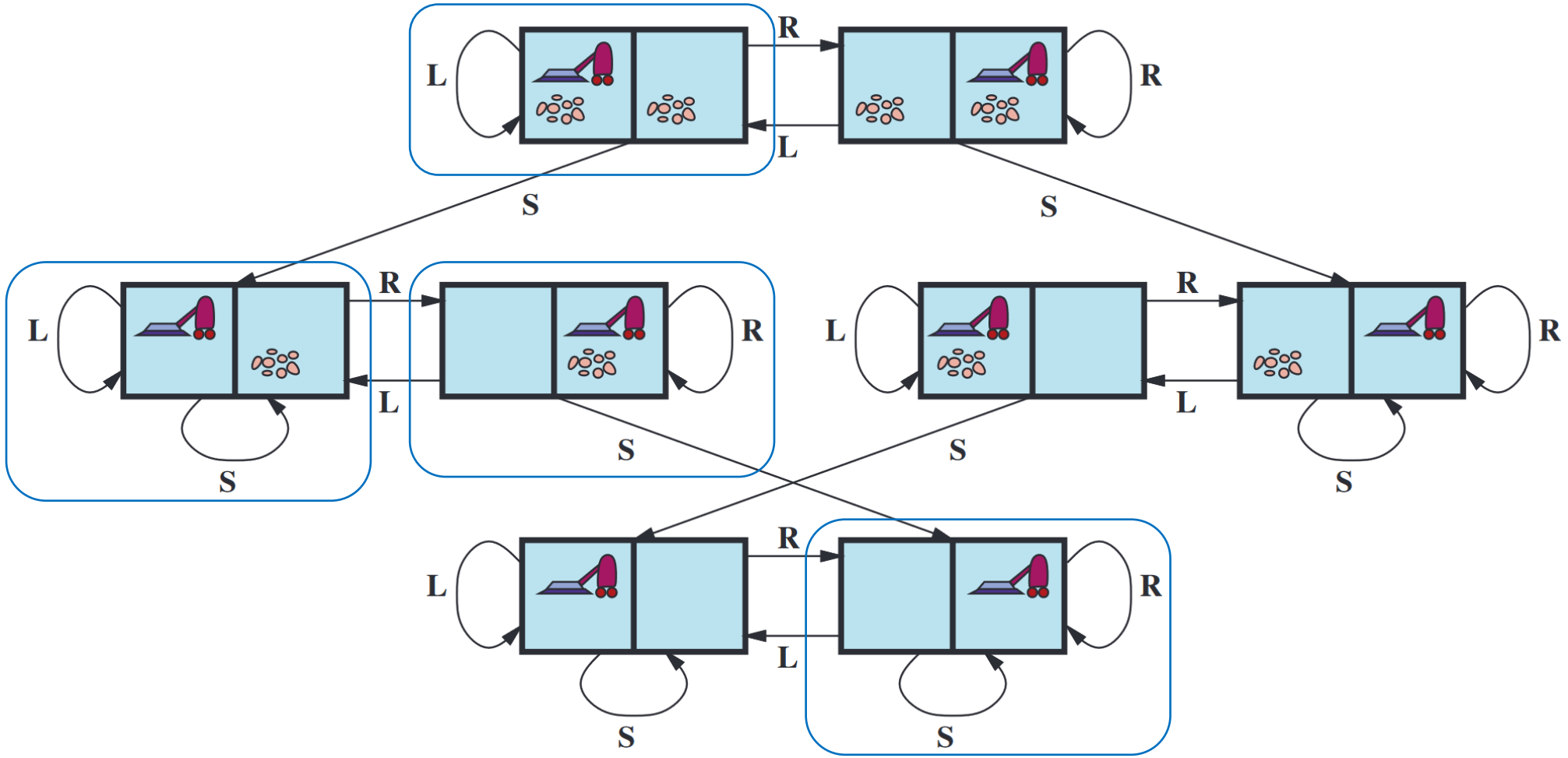
Vacuum World: State Space Graph



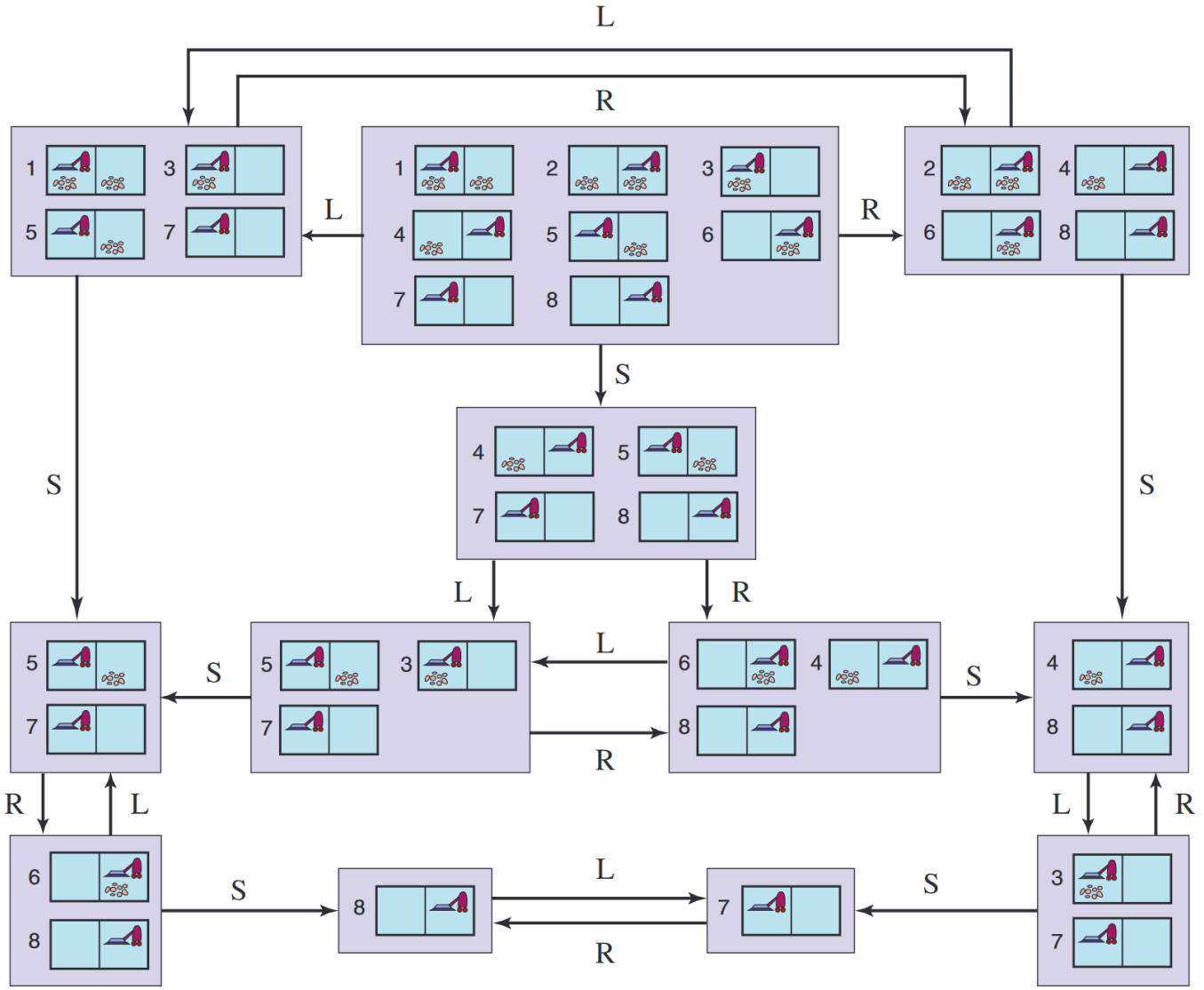
If the environment is completely accessible, the vacuum cleaner always knows where it is and where the dirt is. The solution then can be found by searching for a path from the initial state to the goal state.

States for the search: The world states 1-8.

Solving the Vacuum World



Solving the Vacuum World without Sensors



Problem Formulation: Missionaries and Cannibals

Informal problem description:

- Three missionaries and three cannibals are on one side of a river that they wish to cross.
- A boat is available that can hold at most two people.
- You must never leave a group of missionaries outnumbered by cannibals on the same bank.

➞ How should the state space be represented?
➞ What is the initial state?
➞ What is the goal state?
➞ What are the actions?

One Formalization

State Space: triple (x,y,z) with $0 \leq x,y,z \leq 3$, where x,y, and z represent the number of missionaries, cannibals and boats currently on the original bank.

Initial State: (3,3,1)

Actions: see transition model

Transition Model: from each state, either bring one missionary, one cannibal, two missionaries, two cannibals, or one of each type to the other bank.

Note: not all states are attainable (e.g., (0,0,1)), and some are illegal.

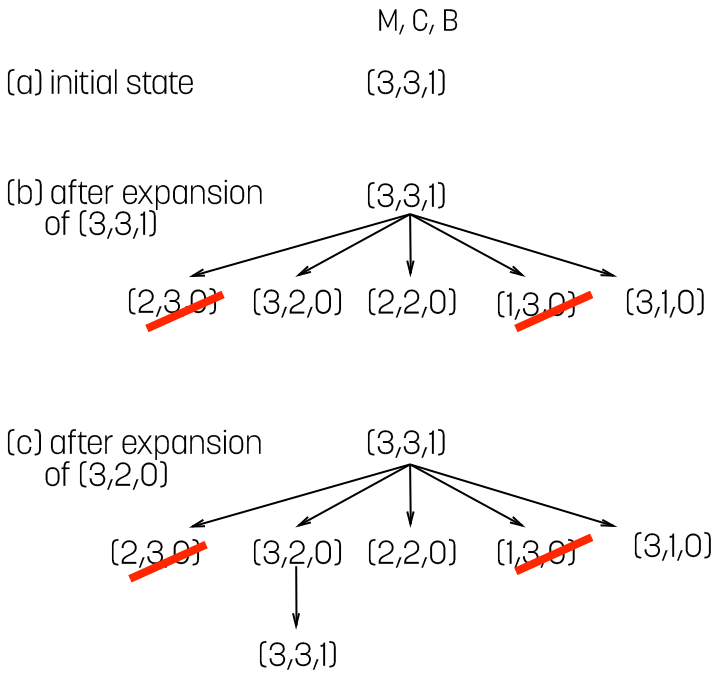
Goal States: {(0,0,0)}

Action Costs: 1 unit per crossing

Many other formalisations

General Search

From the initial state, produce all successive states step by step —> search tree.



Examples of Real-World Problems

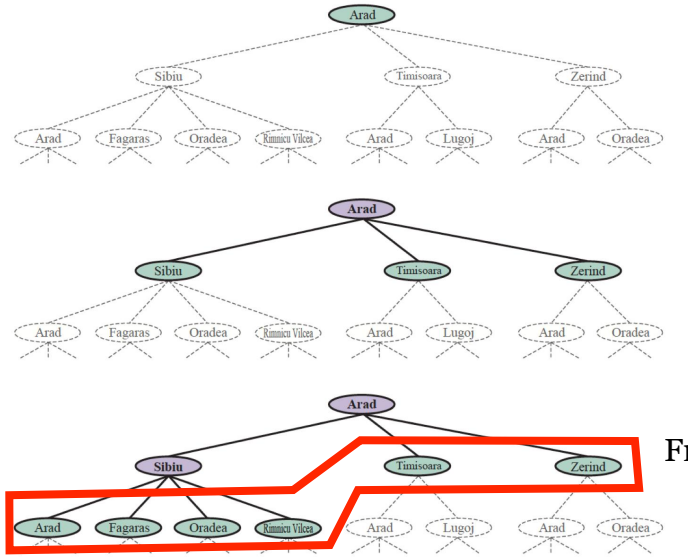
- Route Planning, Shortest Path Problem
  - -Routing video streams in computer networks, airline travel planning, military operations planning…
- Travelling Salesperson Problem (TSP)
  - -A common prototype for NP-complete problems
- VLSI (integrated circuits) Layout
  - -Another NP-complete problem
- Robot Navigation (with high degrees of freedom)
  - -Difficulty increases quickly with the number of degrees of freedom. Further possible complications: errors of perception, unknown environments
- Assembly Sequencing
  - -Planning of the assembly of complex objects (by robots)

Search Algorithms

We focus on algorithms that superimpose a search tree over the state space graph

Important distinction between search tree and state space!

Search tree Construction



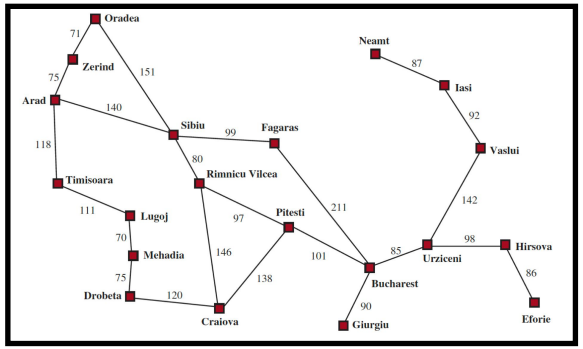
Expanded

Generated

Not Expanded

Frontier (choose from here)

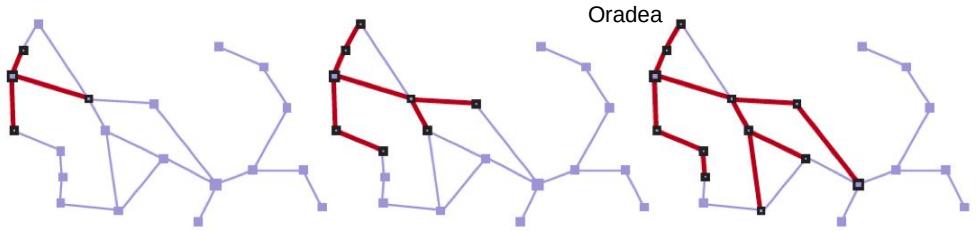
State Space Graph



Graph-Search: Only add a child if the state associated with it has not already been reached:

- Avoid cycles
- Avoid redundant paths

Oradea: has 2 successor states, but already reached by other paths, so do not expand.



Sequence of search trees superimposed (each node only has one parent)

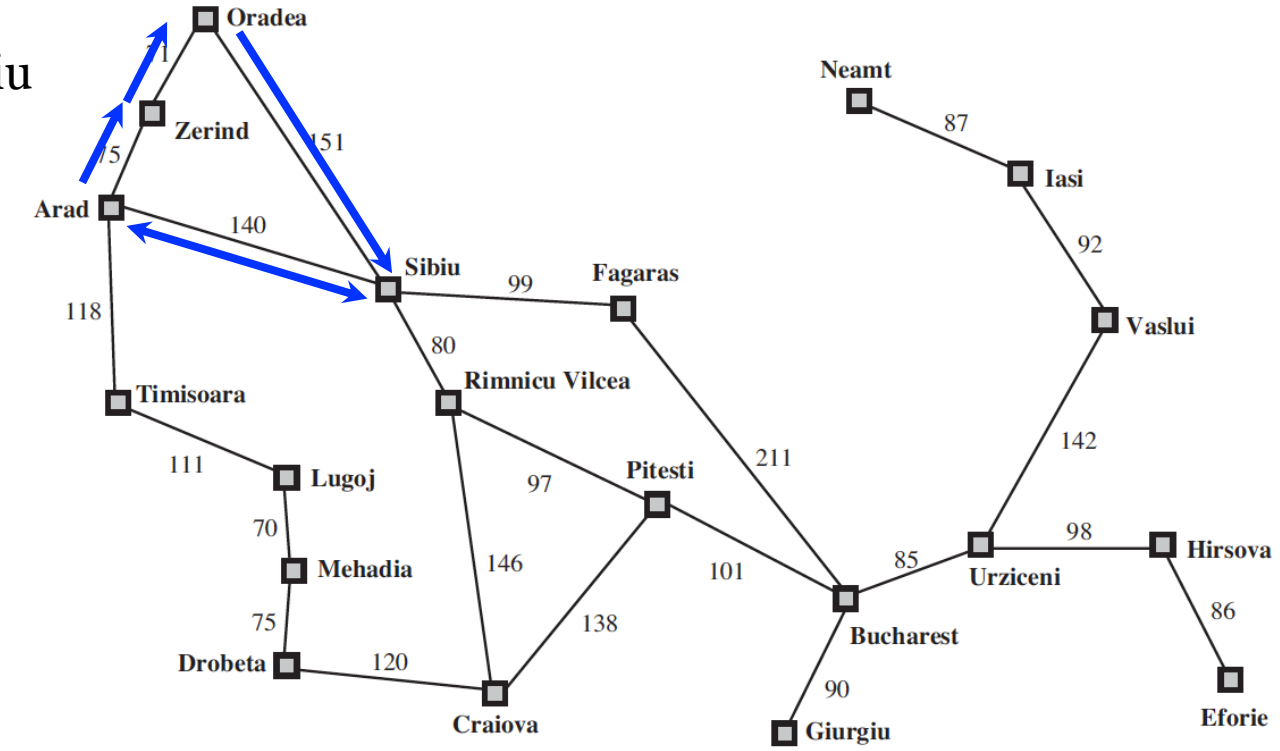
Loopy Paths & Redundant Paths

Arad-Sibiu-Arad

Loopy Path - makes the complete search space infinite even though there are only 20 states

Arad-Sibiu
Arad- Zerind-Oradea-Sibiu

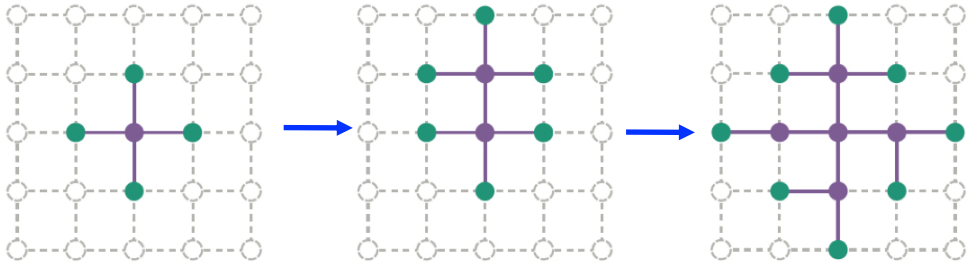
Redundant Path - more than one way to get from one state to another



Cycles are a special case of redundant paths but easier to deal with.

Search Algorithms

Separation property of graph search



The frontier (green) separates the interior (lavender) form the exterior (faint dashed)

Any state with a node generated for it is said to be reached

What is the best way to search through a state space in a systematic manner in order to reach a goal state?

Search Strategies

- A *strategy* is defined by picking the order of node expansion.
- Strategies can be *evaluated* along the following dimensions:
  - *Completeness* – does it find a solution if it exists?
  - *Time Complexity* – number of nodes generated/expanded
  - *Space Complexity* – maximum number of nodes in memory
  - *Optimality* – does it always find a least cost solution
- Time and space complexity are *measured* in terms of:
  - b – *maximum branching* factor of search tree
  - d – *depth of the least cost solution* in the search tree
  - m – *maximum length* of any path in the state space (possibly infinite)

Some Search Classes

- *Uninformed Search* (Blind Search)
  - No additional information about states besides that in the problem definition
  - Can only generate successors and compare against state.
  - Some examples:
    - Breadth-first search, Depth-first search, Iterative deepening DFS
- *Informed Search* (Heuristic Search)
  - Strategies have additional information as to whether non-goal states are more promising than others.
  - Some examples:
    - Greedy Best-First Search, A* Search

Implementing the Search Tree

Data structure for nodes in the search tree:

STATE: state in the state space

PARENT: Predecessor nodes

ACTION: The operator that generated the node

DEPTH: number of steps along the path from the initial state

PATH-COST: Cost of the path from the initial state to the node

Operations on a queue/frontier (4th Edition):

IS-EMPTY(*frontier*): Empty test

POP(*frontier*): Returns the first element of the queue

TOP(*frontier*): Returns the first element

ADD(*node, frontier*): Inserts new elements into the queue

Operations on a queue (3rd Edition):

Make-Queue(Elements): Creates a queue

Empty?(Queue): Empty test

First(Queue): Returns the first element of the queue

Remove-First(Queue): Returns the first element

Insert(Element, Queue): Inserts new elements into the queue

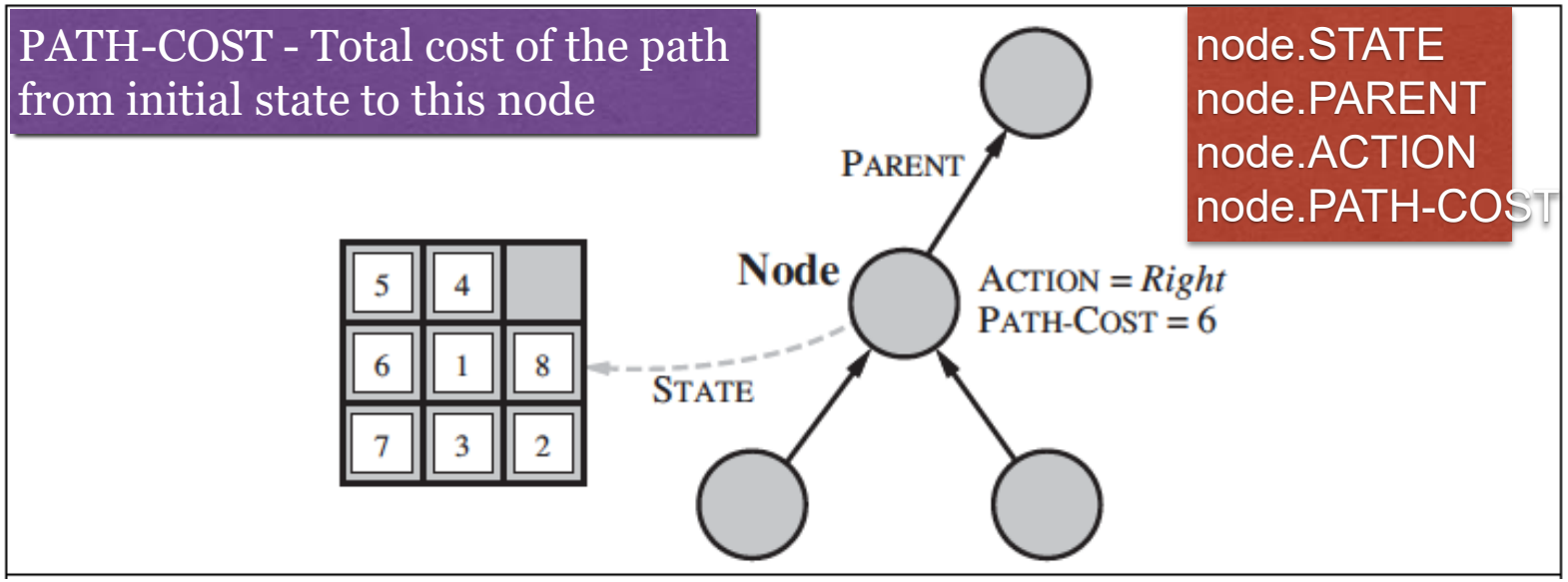
(various possibilities)

Insert-All(Elements, Queue): Inserts a set of elements into the queue

Three kinds of Queues:

- Priority Queue - min cost first
- FIFO Queue - first in, first out
- LIFO Queue - last in, first out

States (in state space) and Nodes (in a search tree)



Can have a finite set of states in state space, but sometimes infinite nodes in a search tree (when loops are allowed)

Can have several nodes with the same state due to multiple paths to the state

The search tree describes paths between states leading towards a goal

Best-First Search

Evaluation function: f(n)

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Reached states

Minimum of f(n) first

Bookkeeping for dealing with loopy and redundant paths

Goal test during selection for expansion rather than at node generation

Expand node, generate children

Place child in reached if not there already, or if there but path cost is cheaper, then add to frontier

Yields sequence of child nodes

Different evaluation functions f(n), will generate different algorithms

Uniform-Cost Search

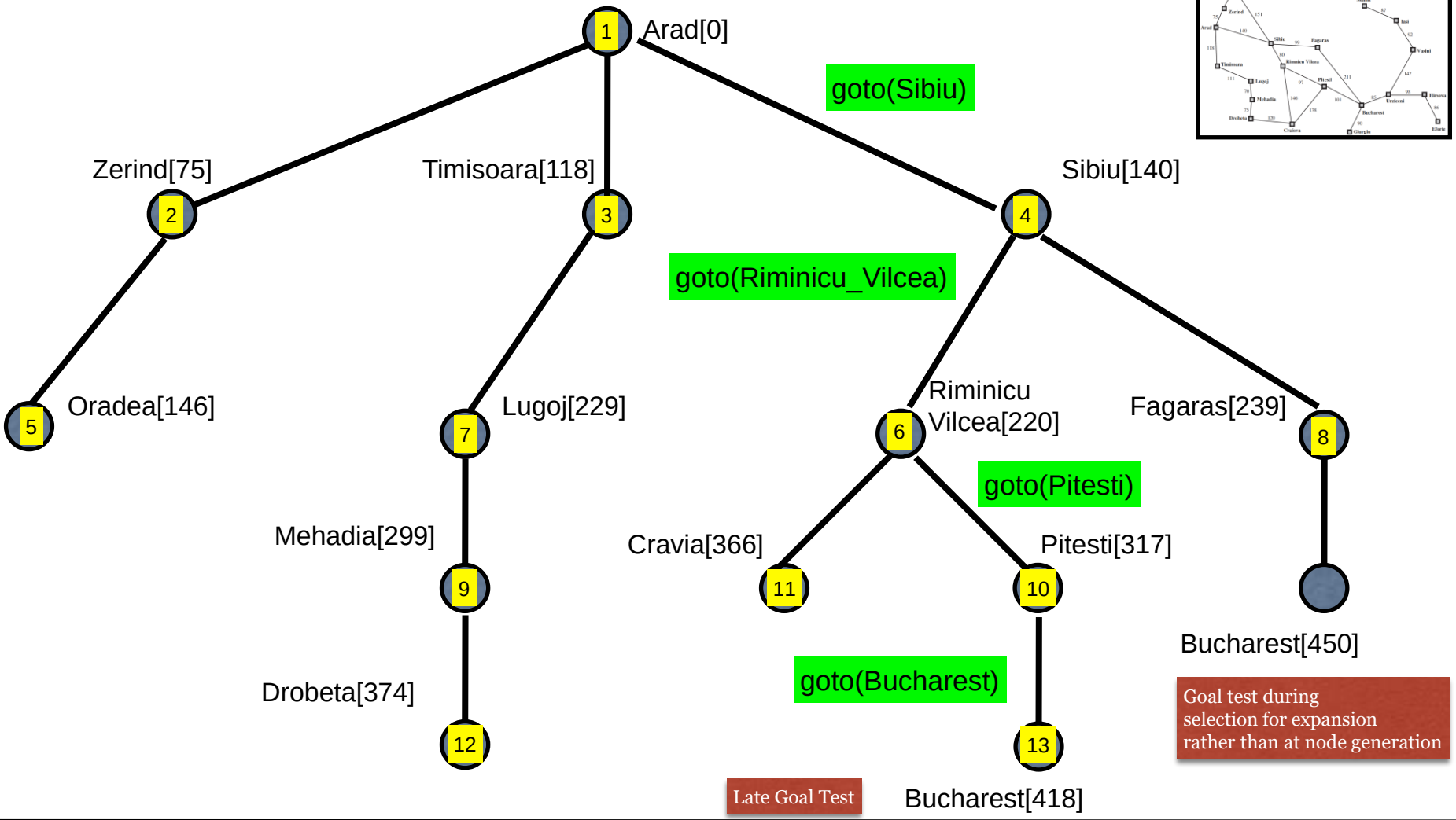
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
return BEST-FIRST-SEARCH(*problem*, PATH-COST)

When actions have *different* costs, BEST-FIRST SEARCH can be used where the cost-function f(n) = n.PATH-COST

Computer Science: Djikstra's Algorithm
Artificial Intelligence: Uniform Cost Search

Search tree spreads out in waves of uniform cost

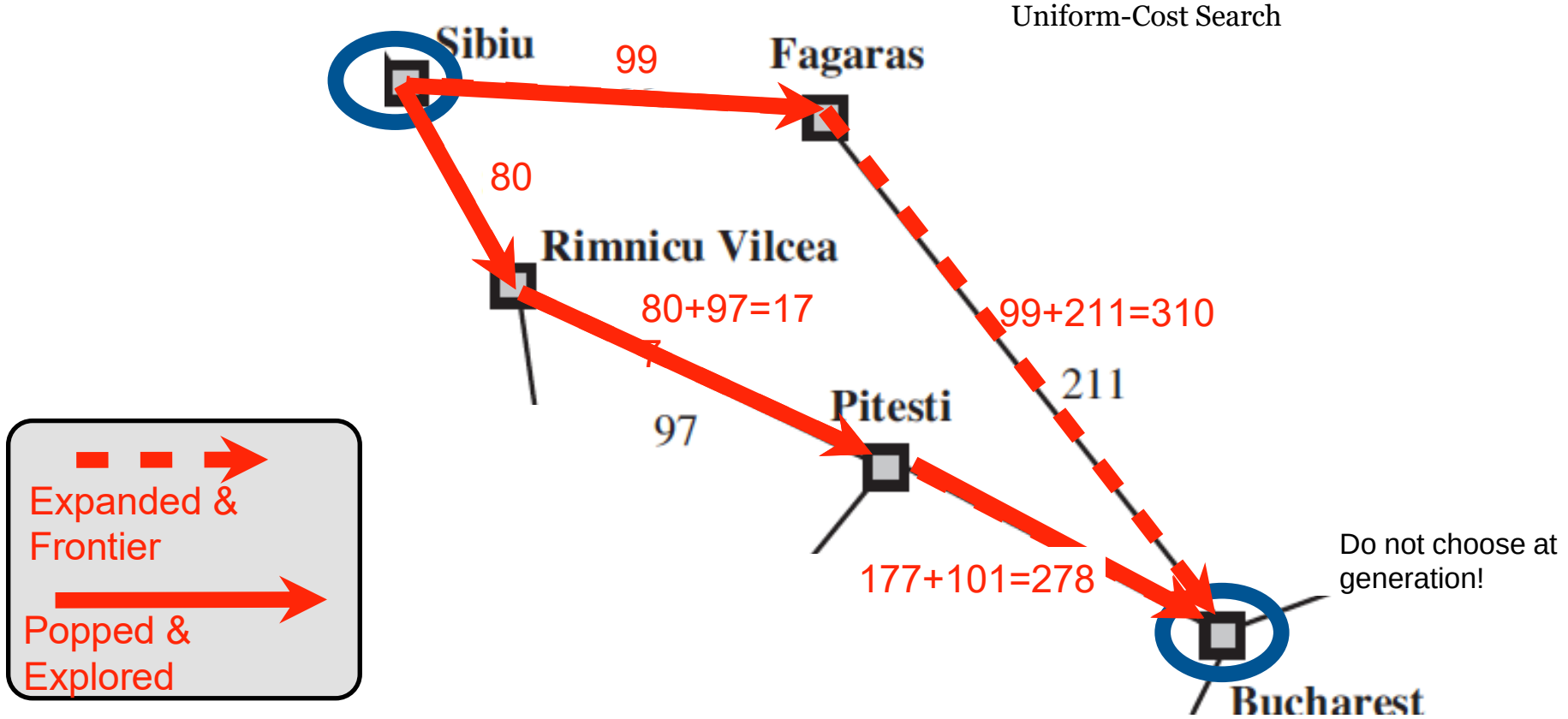
Uniform-Cost Search: Romania



- Cost-optimal: 1st solution found has cost at least as low as any in the frontier
- Complete: Systematically checks paths in order of increasing cost

Early Goal Test / Late Goal Test

- Early Goal Test: Test during node generation
- Late Goal Test: Test when popped off the queue (best-first search)



Late Goal Test!

Redundant Paths Re-visited

- A search algorithm is called a graph search if it it checks for redundant paths
- A search algorithm is called a tree-like search if if it does not check for redundant paths.
  - *Tree-like* because the state space is still the same graph. We just choose to treat it as if it is a tree.

3 approaches to redundant paths:

1. Remember all previously reached states, allowing detection of redundant paths, keep only the best path to each state.
2. Do not worry about repeating the past. Don't track reached states and don't check for redundant paths
3. Compromise and just check for cycles (loopy paths). Just follow chain of parents up a path. No need for additional memory.

Breadth-First Search

Assume all actions have the same cost

Search by Minimal Depth:

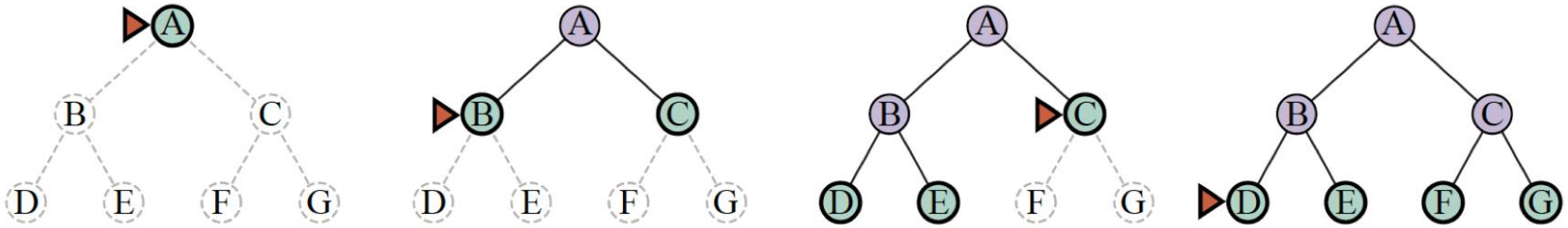


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Could be implemented using Best-First-Search:

function BREADTH-FIRST-SEARCH *problem*) **returns** a solution node, or *failure*
return BEST-FIRST-SEARCH(*problem*, DEPTH)

f(n) = n.DEPTH

Is a better way!

Breadth-First Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

FIFO Queue

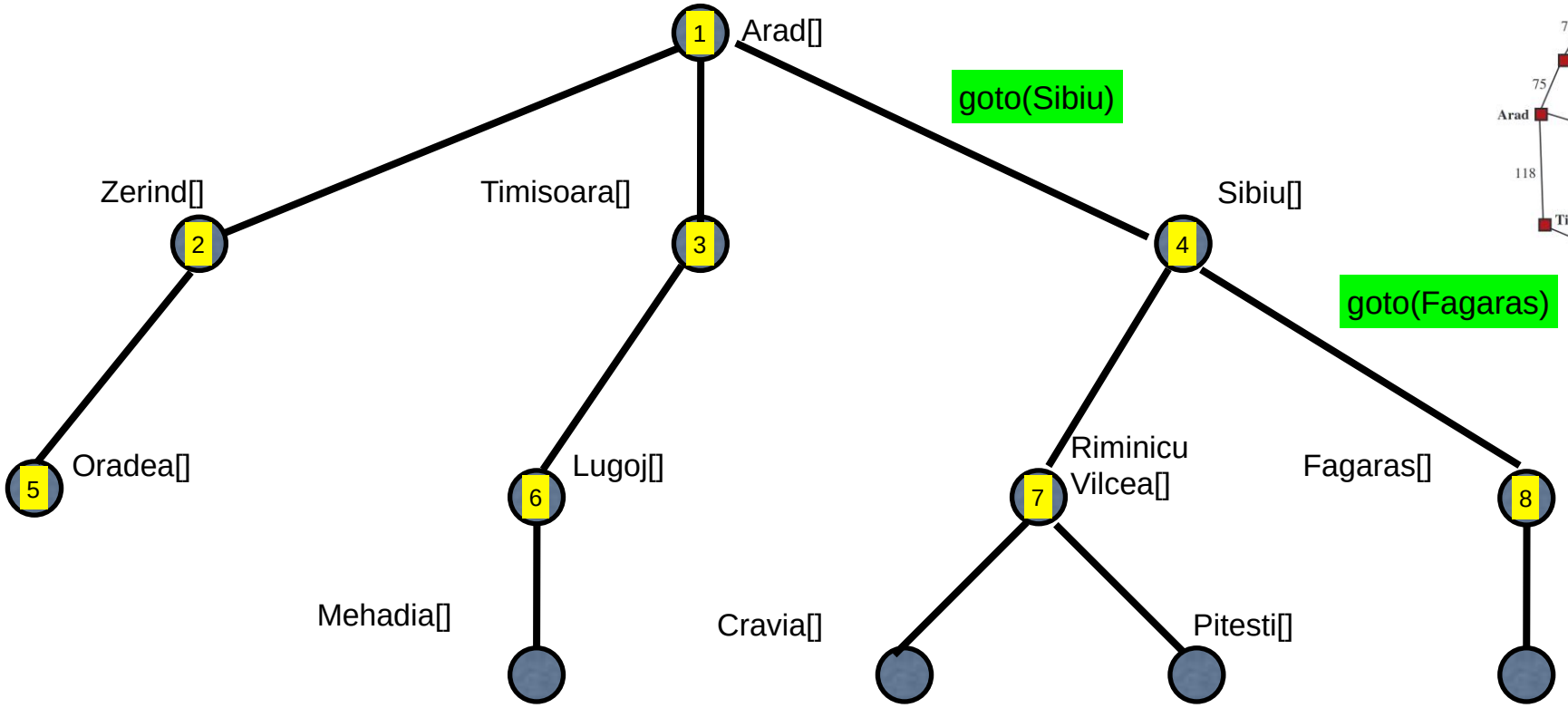
Reached is a set of states

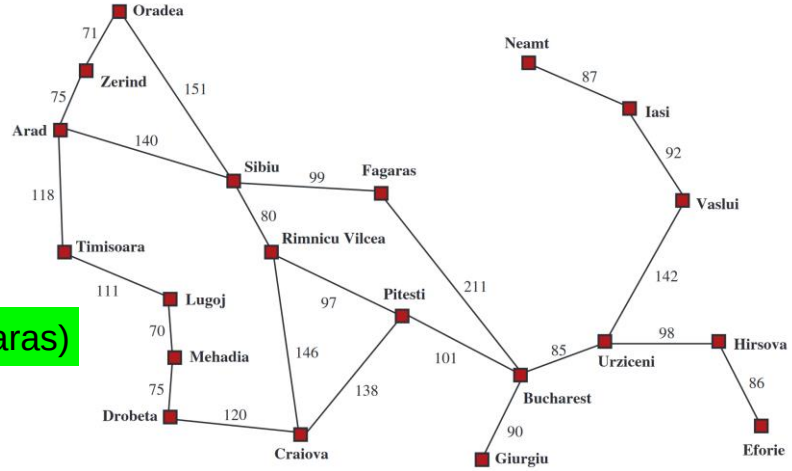
Early Goal Test

Optimizations:

- FIFO queue is faster than a priority queue
- *Reached* is a set of states rather than a mapping from states to nodes
- Can do an Early Goal Test when node is generated
  - Once a state is reached, can never find a better path to the state

Breadth-First Search: Romania





- Cost-optimal for problems where all actions have same cost
- Finds solution with minimal number of actions
- It is complete

Depth-First Search

Assume all actions have the same cost

Always expands
deepest node
in the frontier first

Usually implemented not as
graph search but as
tree-like search (without a
table of reached states)

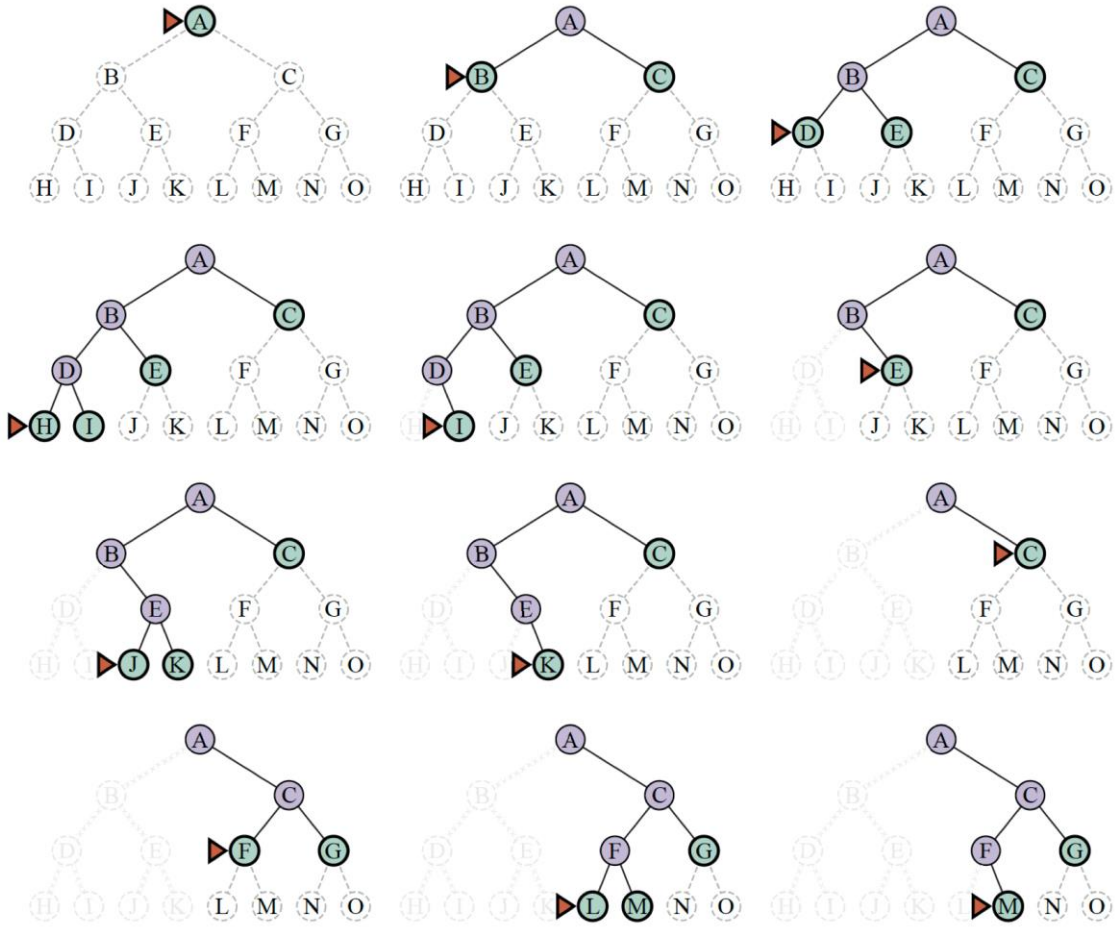


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

Depth-First Search

Could be implemented using Best-First-Search:

function DEPTH-FIRST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, NEGDEPTH)

$$f(n) = \text{NEGDEPTH} = -\ n.\text{DEPTH}$$

There is a Better way!

Depth-First Search (Graph Search)

```
function DEPTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a LIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

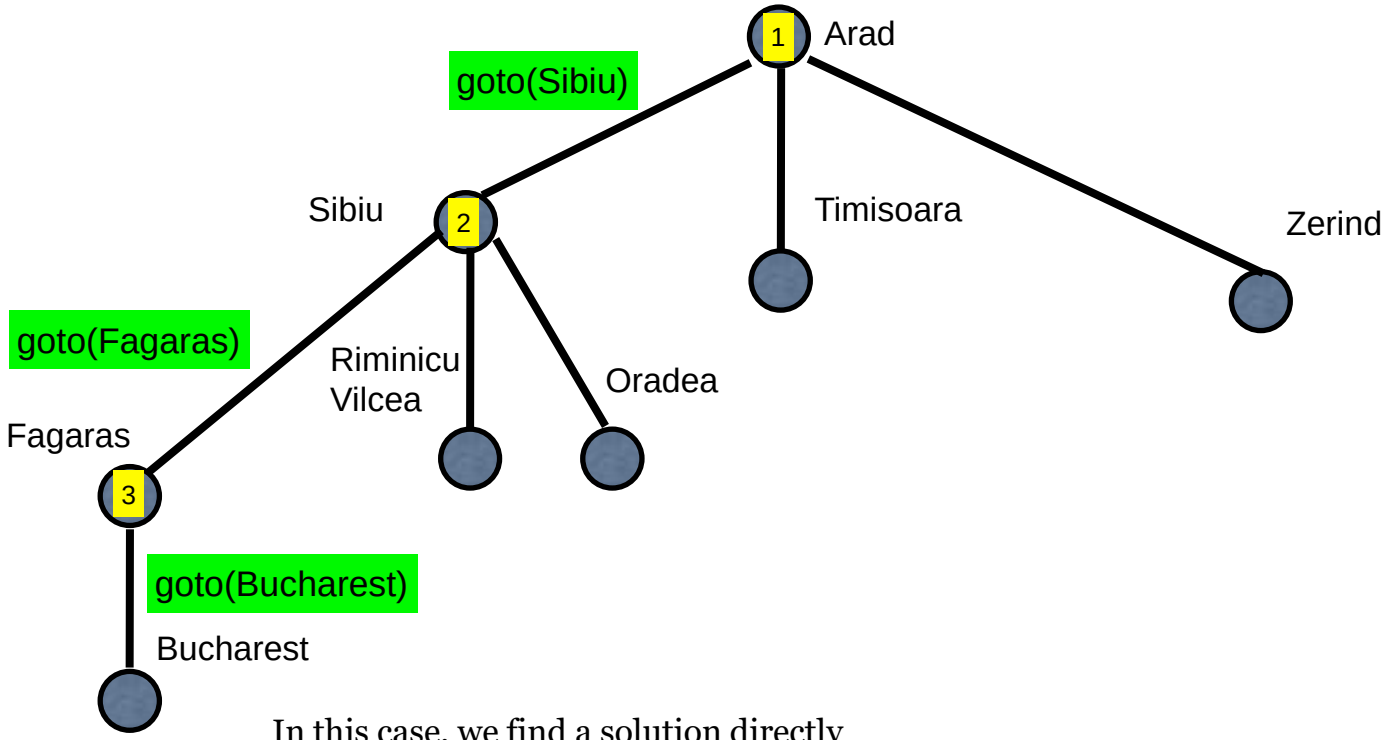
LIFO Queue

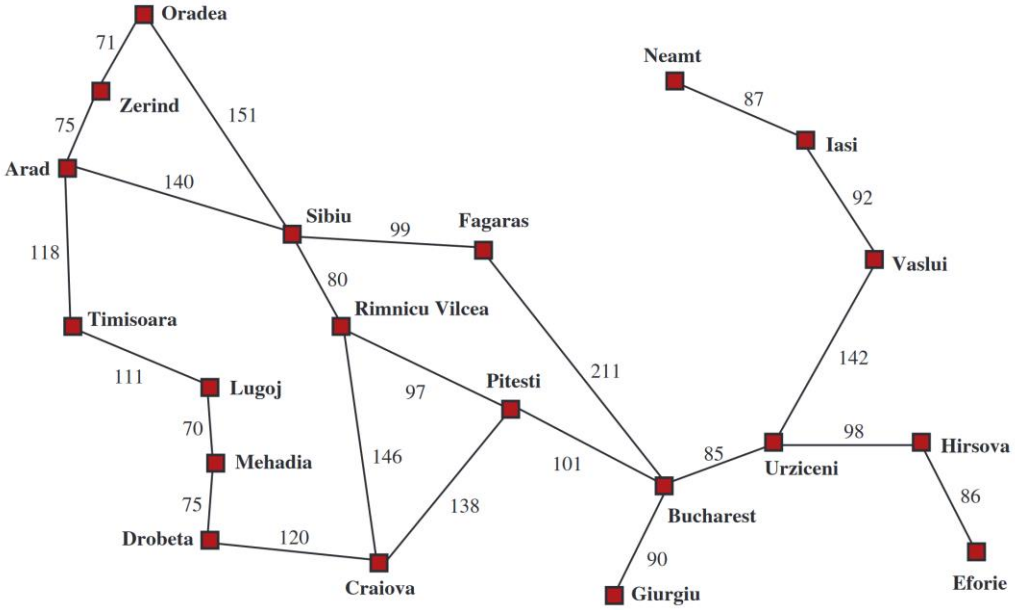
Early Goal Test

Optimizations:

- LIFO queue is faster than a priority queue
- Can do an Early Goal Test when node is generated

Depth-First Search (Graph Search): Romania





In this case, we find a solution directly

Depth-First Search (Tree-like Search)

```
function DEPTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a LIFO queue, with node as an element
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      add child to frontier
  return failure
```

LIFO Queue

No bookkeeping of reached nodes

Early Goal Test

Note, although it is tree-like search, a cycle check could be added and often is

But:

- DFS (both graph and tree-like) is not cost-optimal
- For finite state spaces that are trees: Complete
- For cyclic state spaces, can get stuck in infinite loop
- For infinite state spaces, not systematic, can get stuck in an infinite path
- *Bottom line*: not cost-optimal, not complete in general

But:

- For tree-like search uses very small amount of memory
- Because of this, it is the basic workhorse of many areas of AI
- In particular: DFS with backtracking uses even less memory

Depth-Limited Search (Tree-like Search)

- Deals with failure of depth-first search in infinite state spaces
- Introduce a pre-determined cut-off depth limit l

```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Cycle-check: can be full cycle check up to *l* or just *k<l*

If $l < d$ then we may not find the goal and DLS is incomplete
If $l > d$ then DLS is not optimal
DLS with $l = \infty$ is in fact depth-first search

Sometimes a good depth-limit can be chosen based on the problem

Iterative-Deepening Search

- Combines the best of depth-first search and breadth-first search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

- Gradually increases the depth-limit of depth-first search by increments (0,1,2…).
- Each increment basically does a breadth-first search to that limit

Complete when the branching factor is finite

Optimal when the path cost is a non-decreasing function of the depth of the node

In general, Iterative-Deepening is the preferred uninformed search method when the state space is larger than can fit in memory and the solution depth is unknown

Iterative-Deepening Search: Example

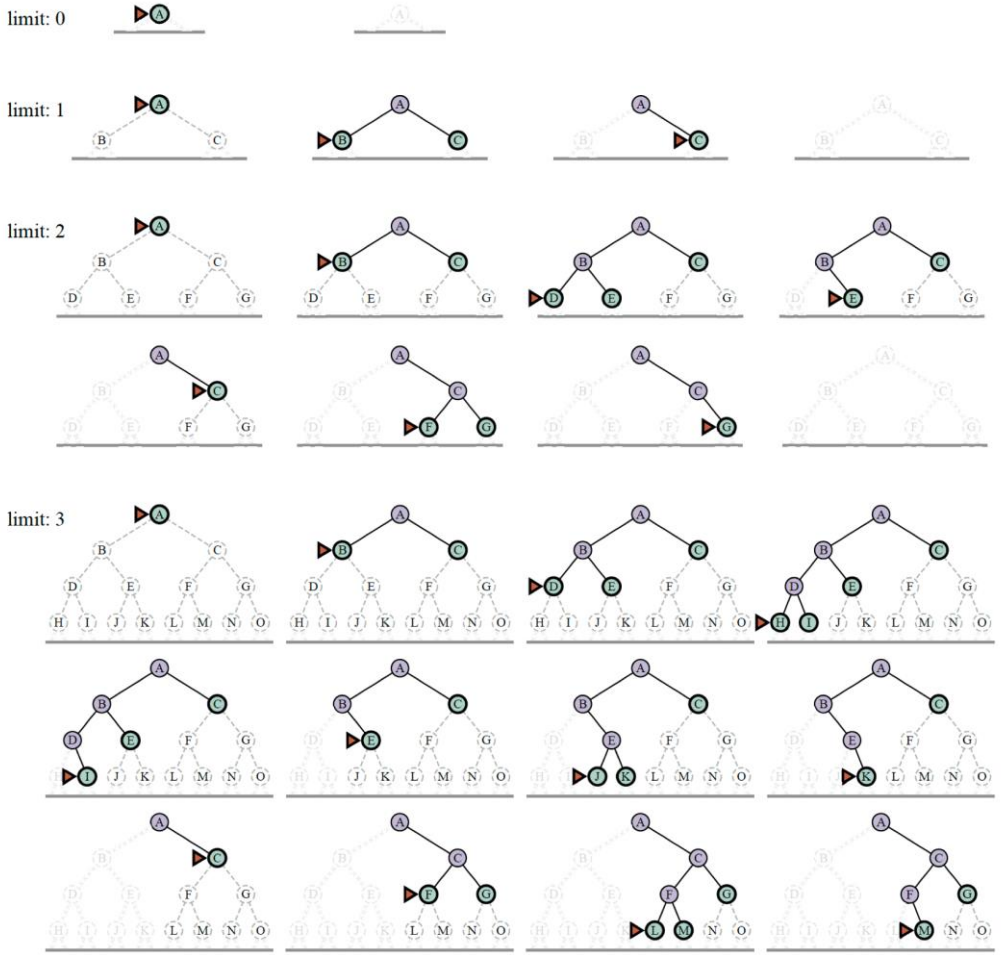
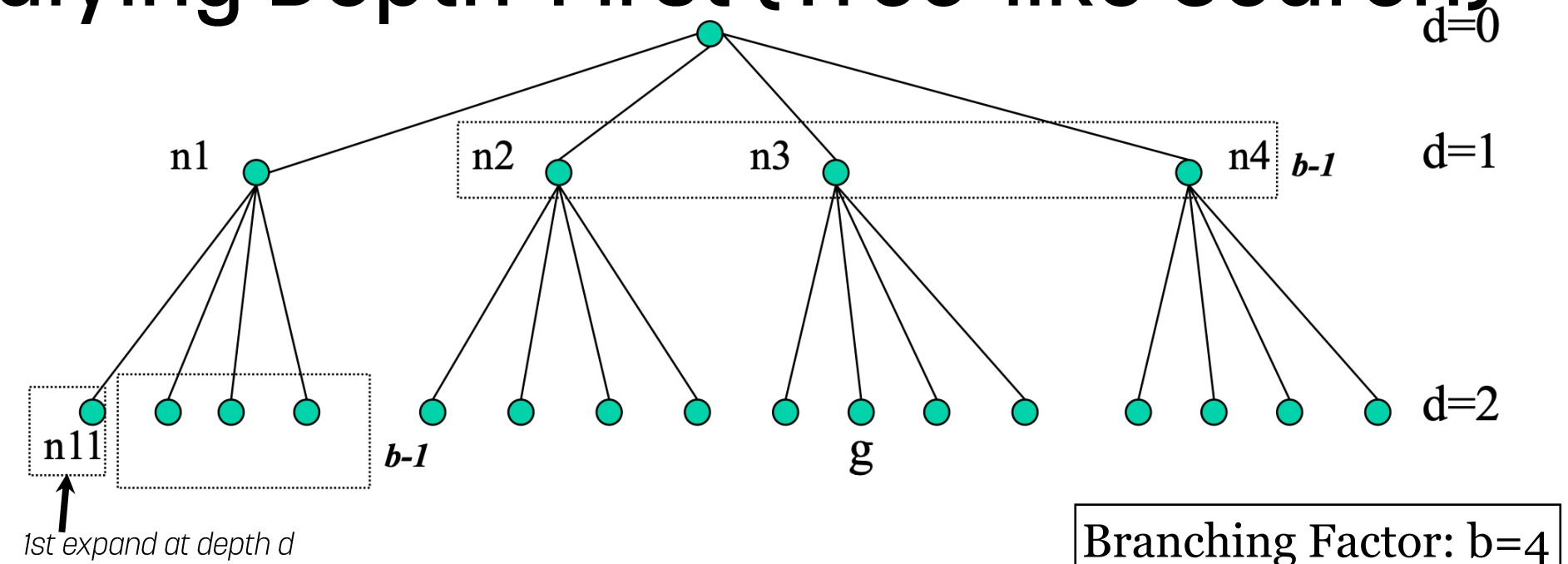


Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

Analying Depth-First (Tree-like Search)

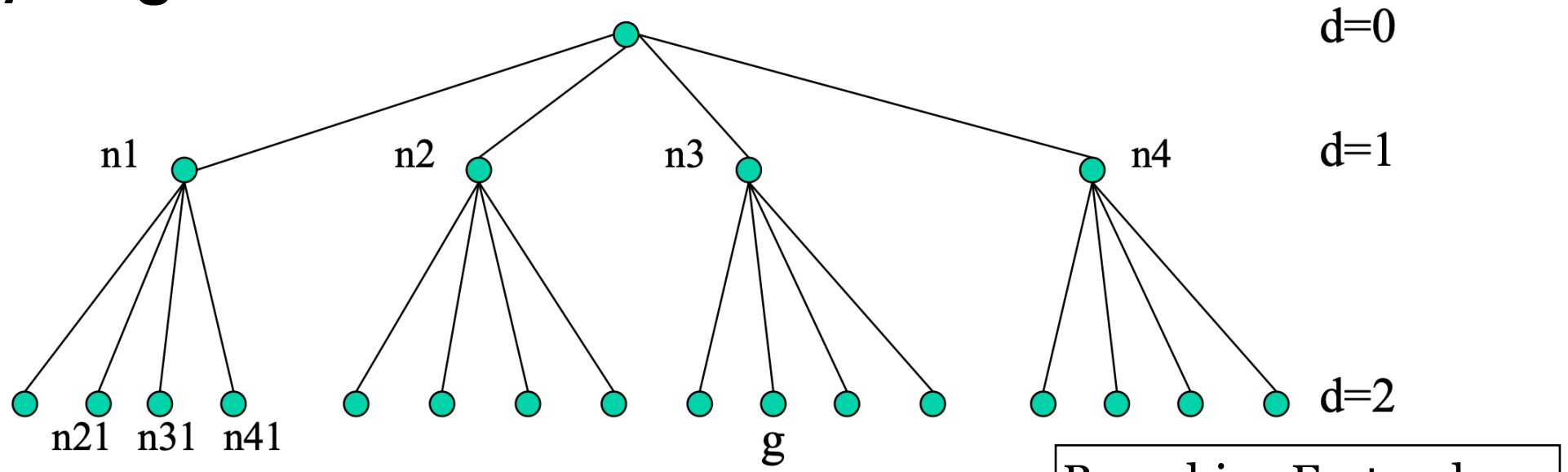


Branching Factor: b=4
Goal Depth: d=2

- For any tree-like search, when checking for a goal node at level d, at most d(b-1) nodes must be stored in the frontier.

Space Complexity: $O(d(b-1))$

Analysing Breadth-First Search (Tree-Like Search)



- For any graph search, every expanded node is stored in the reached set.
- For tree-like search: ***O***(b^d) nodes in the frontier set.
- The space complexity is dominated by the nodes in the frontier.

Space Complexity: $O(b^d)$

Uninformed Search Algorithms: Comparison

Tree-Like Search Versions

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

m -max length of any path in the state space

For Graph-Search Versions:

- DFS is complete for Finite Spaces
- All algorithms: Space and Time complexities bounded by the size of the state space: |Vertices| + |Edges|

TDDC17 AI LE2 HT2023:
Physical Symbol Systems
Heuristic Search Hypothesis
Search I: Uninformed Search Algorithms (Ch 3)

www.ida.liu.se/~TDDC17

LINKÖPING UNIVERSITY