Lecture 4

CS 6380 Artificial Intelligence

Uninformed Search

Department of Computer Science and Engineering

Indian Institute of Technology Madras (IITM), India

Some material based on lecture slides for CSE 571: Artificial Intelligence by Siddharth Srivastava, ASU.

Logistics and Recap

Resources

- Course Website: https://hails-group.github.io/courses/cs6380/jn26.html
  - Slides posted here
  - Schedule already available

- Announcements: Moodle

- Course Email: pulkitv@cse.iitm.ac.in
  - Subject: [CS6380] <Subject>
  - No URGENT email responses. Please **plan** ahead.
  - We will respond to each email within 48 hours.
  - Any course-related query should be posted on Moodle so that others can also benefit from the answer.

Exams

- 3 Exams:
  1. [Quiz 1] Aug 31
  2. [Quiz 2] Oct 12
  3. [End Sem] Nov 11

- More details closer to exams.

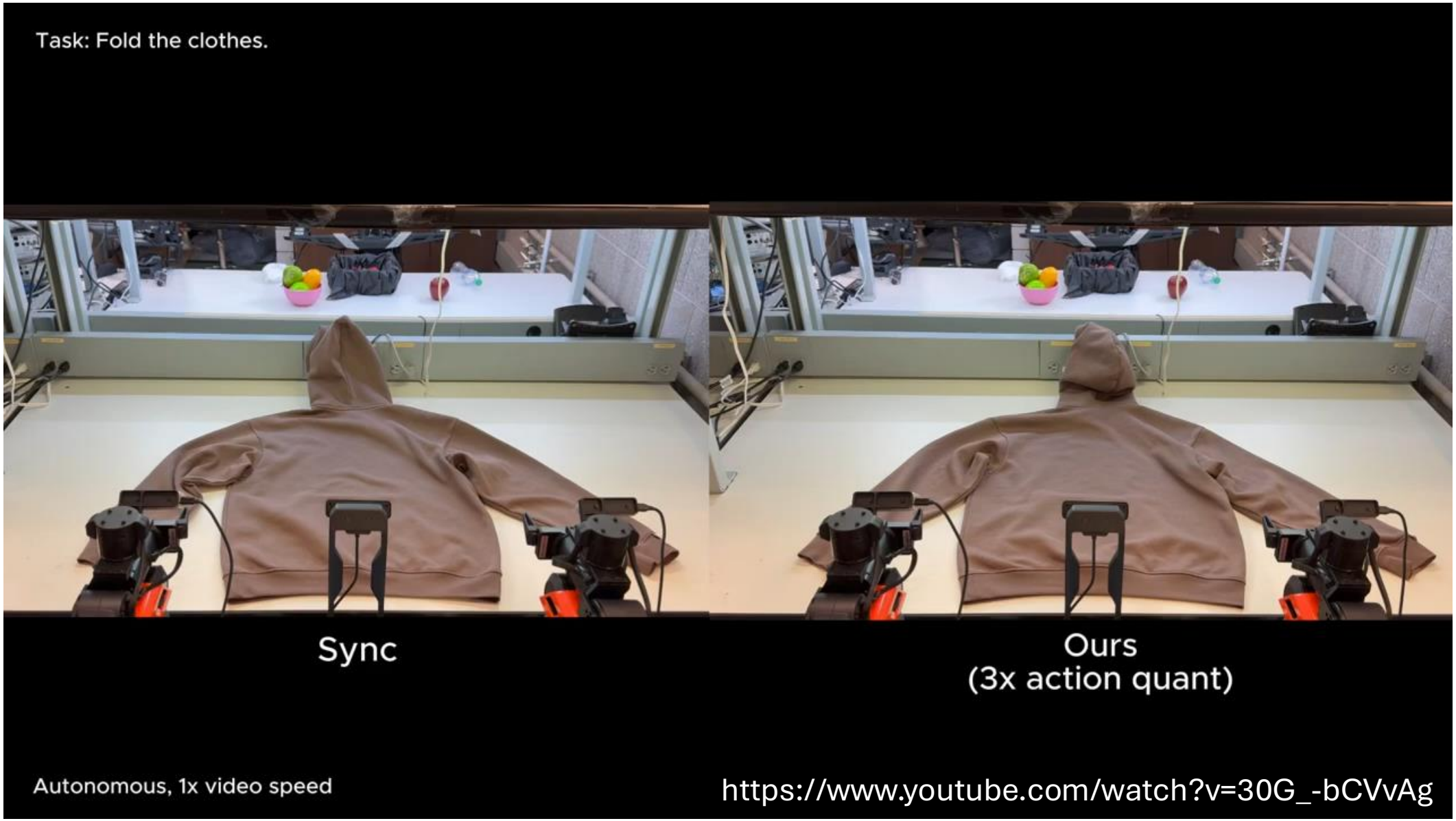
Grading Breakup

- Exams: 70%
  1. Quiz 1 : 20%
  2. Quiz 2 : 20%
  3. End Sem: 30%
- Assignment-Quizzes: 25%
- In-class Quizzes: 5%
- Relative Grading

Announcement

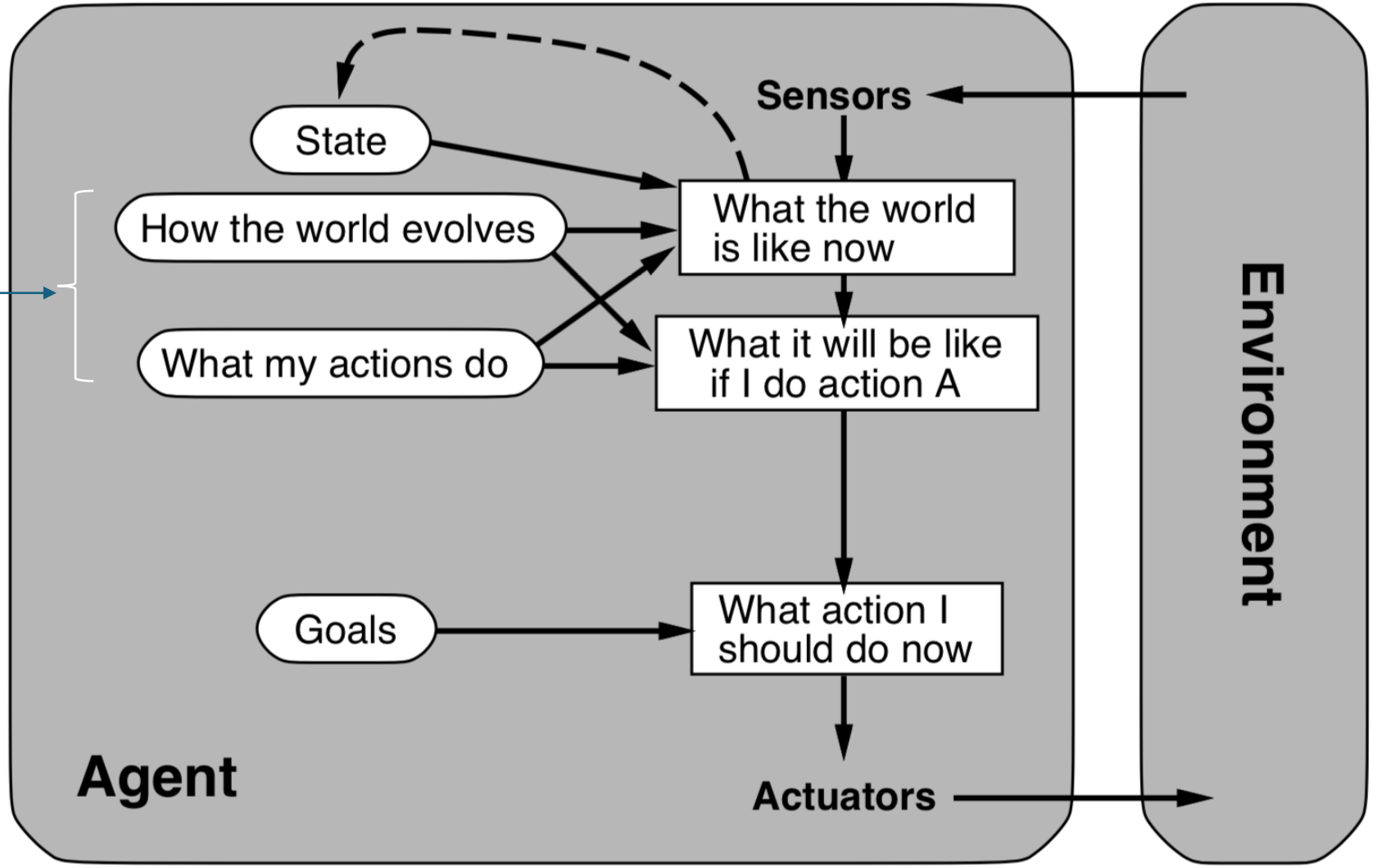
- No class tomorrow (Aug 4)

Advances in Laundry Robots



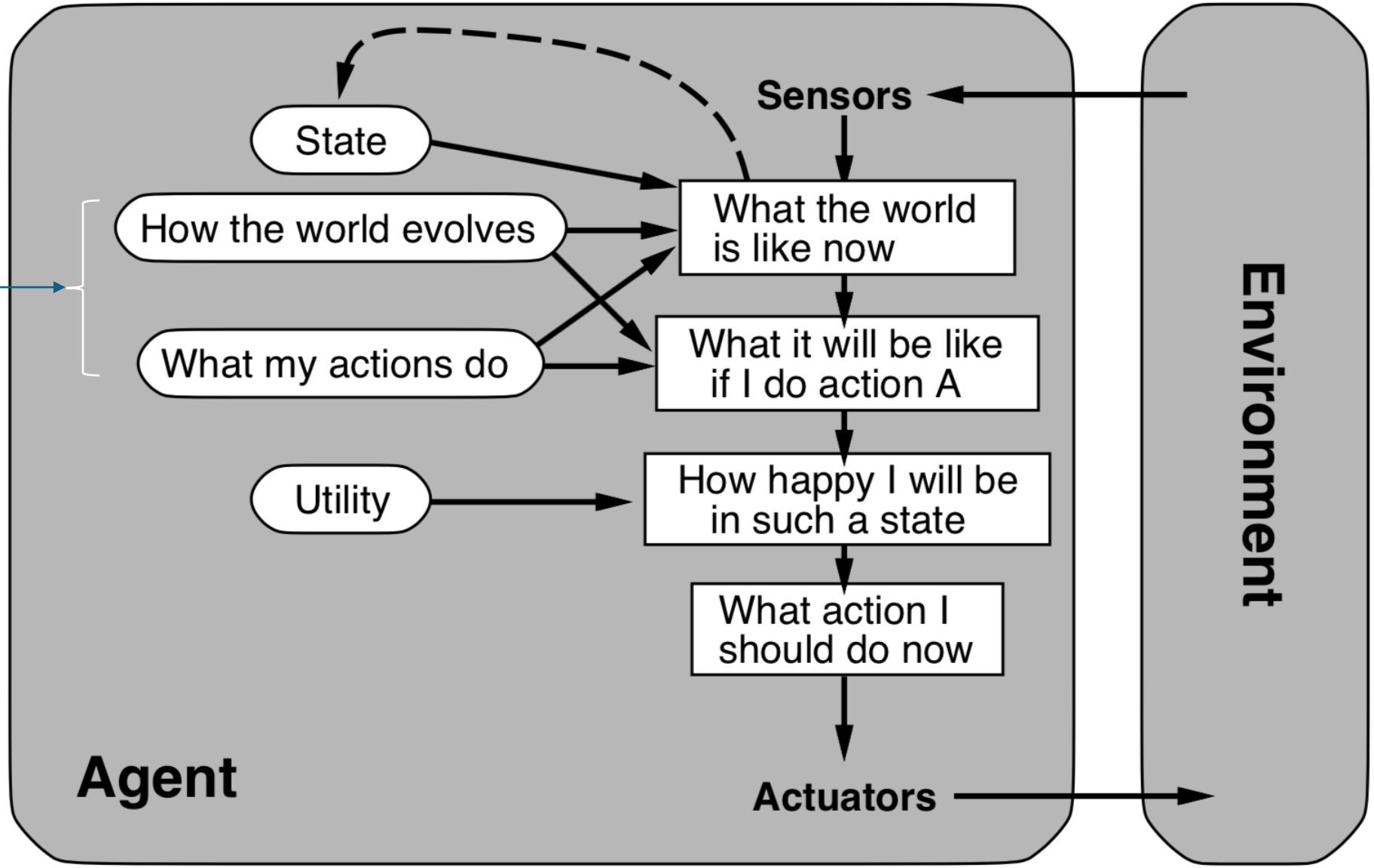
Goal-based Agents

Models:
Simulators or
Analytical specs

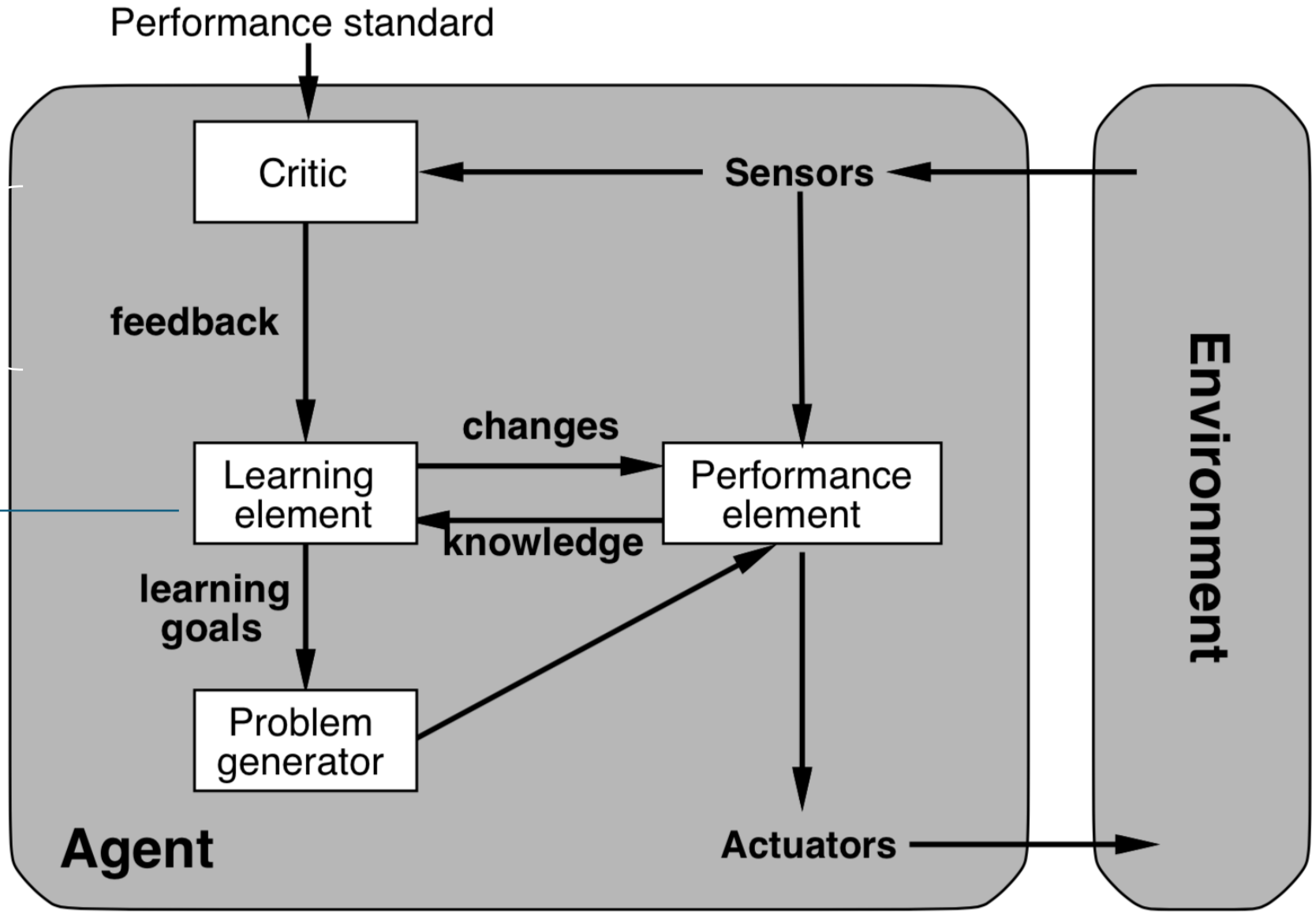


Utility-based Agents

Models:
Simulators or
Analytical specs

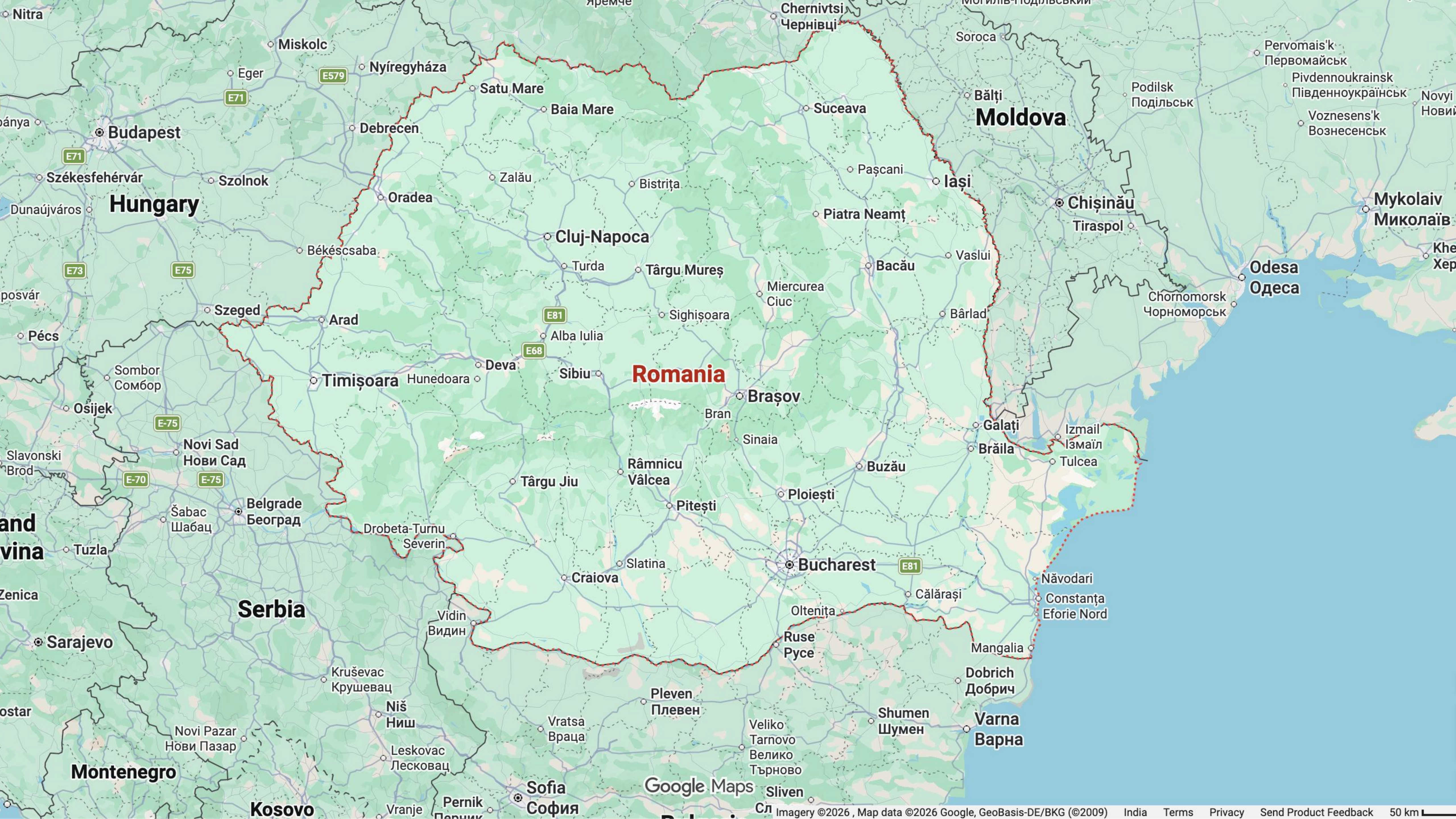


Learning Agents



May learn or modify models

Problem Formulation

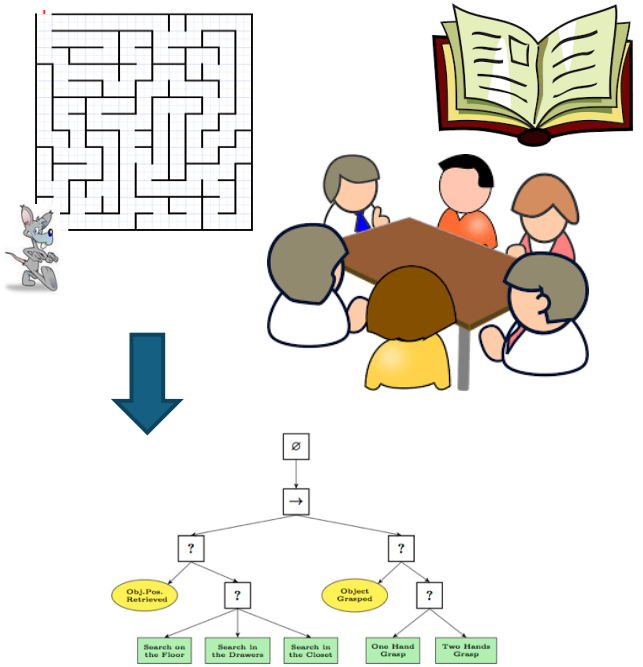

Romania
Hungary
Moldova
Serbia
Montenegro
Kosovo
Budapest
Nitra
Miskolc
Eger
Nyíregyháza
Debrecen
Satu Mare
Baia Mare
Székesfehérvár
Szolnok
Dunaújváros
Békéscsaba
Szeged
Pécs
Sombor
Сомбор
Osijek
Novi Sad
Нови Сад
Slavonski
Brod
Belgrade
Београд
Šabac
Шабац
Tuzla
Zenica
Sarajevo
Kruševac
Крушевац
Niš
Ниш
Novi Pazar
Нови Пазар
Leskovac
Лесковац
Vranje
Pernik
Sofia
София
Vidin
Видин
Vratsa
Враца
Pleven
Плевен
Veliko
Tarnovo
Велико
Търново
Sliven
Shumen
Шумен
Varna
Варна
Dobrich
Добрич
Ruse
Русе
Oradea
Zalău
Bistrița
Cluj-Napoca
Turda
Târgu Mureș
Arad
Timișoara
Alba Iulia
Deva
Hunedoara
Sibiu
Sighișoara
Miercurea
Ciuc
Brașov
Bran
Sinaia
Râmnicu
Vâlcea
Târgu Jiu
Pitești
Ploiești
Buzău
Drobeta-Turnu
Severin
Craiova
Slatina
Bucharest
Oltenița
Călărași
Suceava
Pașcani
Iași
Piatra Neamț
Bacău
Vaslui
Bârlad
Galați
Brăila
Tulcea
Izmail
Ізмаїл
Năvodari
Constanța
Eforie Nord
Mangalia
Chernivtsi
Чернівці
Soroca
Bălți
Chișinău
Tiraspol
Podilsk
Подільськ
Pervomais'k
Первомайськ
Pivdennoukrainsk
Південноукраїнськ
Voznesens'k
Вознесенськ
Mykolaiv
Миколаїв
Odesa
Одеса
Chornomorsk
Чорноморськ
E579
E71
E73
E75
E81
E68
E-75
E-70
Google Maps
Imagery ©2026 , Map data ©2026 Google, GeoBasis-DE/BKG (©2009)
India
Terms
Privacy
Send Product Feedback
50 km

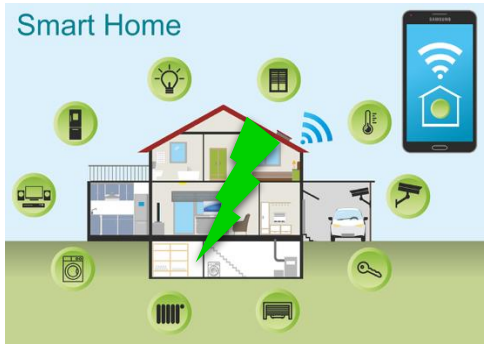
The Typical Agent Design Process

1. **Represent** background information and objective
   - Multi-step agents require states, actions and their effects on states
   - Learned using simulator, modeled using literature/experts, or partially modeled with scope for online learning

2. **Compute** "behavior" that would allow agent to achieve objective
   - Behavior may include contingencies depending on observed system state during execution

3. **Execute** this behavior
   - The bottomline, drives all of the above





How Agents Represent the World

- Every component of an agent must represent the environment somehow. Three levels, in increasing order of expressiveness:

- **Atomic**: a state is a black box with no internal structure
  - The only operation available is: are these two states identical?
  - Used by search, game-playing, hidden Markov models, Markov decision processes
- **Factored**: a state is a fixed set of variables, each with a value
  - Two different states can now share some values, and you can say what changed
  - Used by constraint satisfaction, propositional logic, planning, Bayesian networks, most machine learning
- **Structured**: a state contains objects, each with attributes and relationships to other objects
  - Used by relational databases, first-order logic, first-order probability models, natural language understanding

- **The trade-off:** more expressive representations describe complex worlds far more concisely, but reasoning with them is correspondingly harder.

Representation Drives the Rest of the Course

- **Atomic representations**
  - Uninformed and informed search; search in complex environments; adversarial search and games
  - Also: Markov decision processes and reinforcement learning

- **Factored representations**
  - Constraint satisfaction problems; propositional logic and satisfiability
  - Classical planning; probabilistic reasoning and Bayesian networks; most of machine learning

- **Structured representations**
  - First-order logic and inference; knowledge representation
  - Relational probability models; natural language understanding

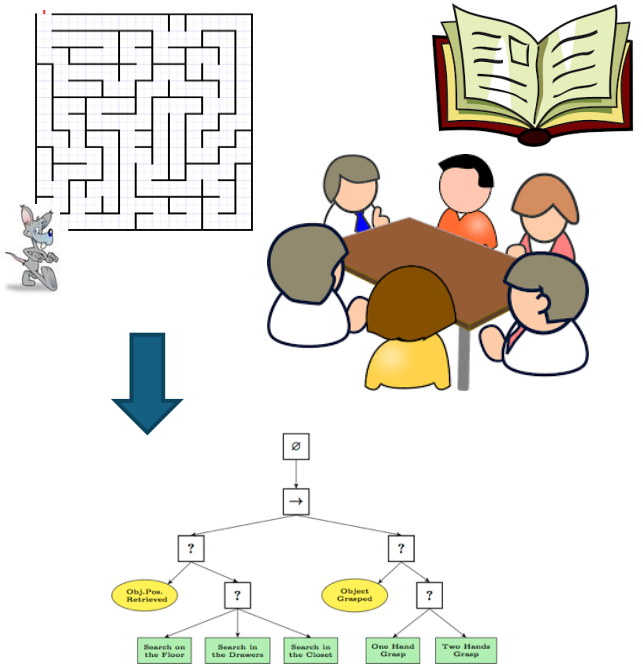
- **How to use this when you design an agent**
  - Choose the **coarsest** representation that still captures what your performance measure depends on
  - Every step up in expressiveness costs you computationally; pay for it only when the problem demands it

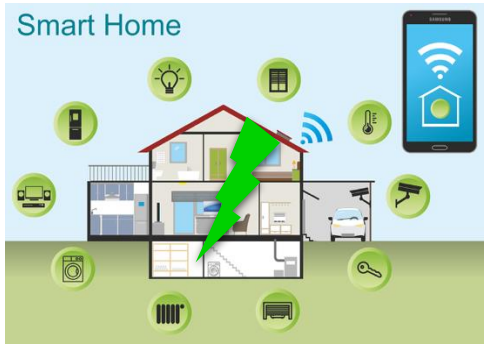
The Typical Agent Design Process

1. **Represent** background information and objective
   - Multi-step agents require states, actions and their effects on states
   - Learned using simulator, modeled using literature/experts, or partially modeled with scope for online learning

2. **Compute** "behavior" that would allow agent to achieve objective
   - Behavior may include contingencies depending on observed system state during execution

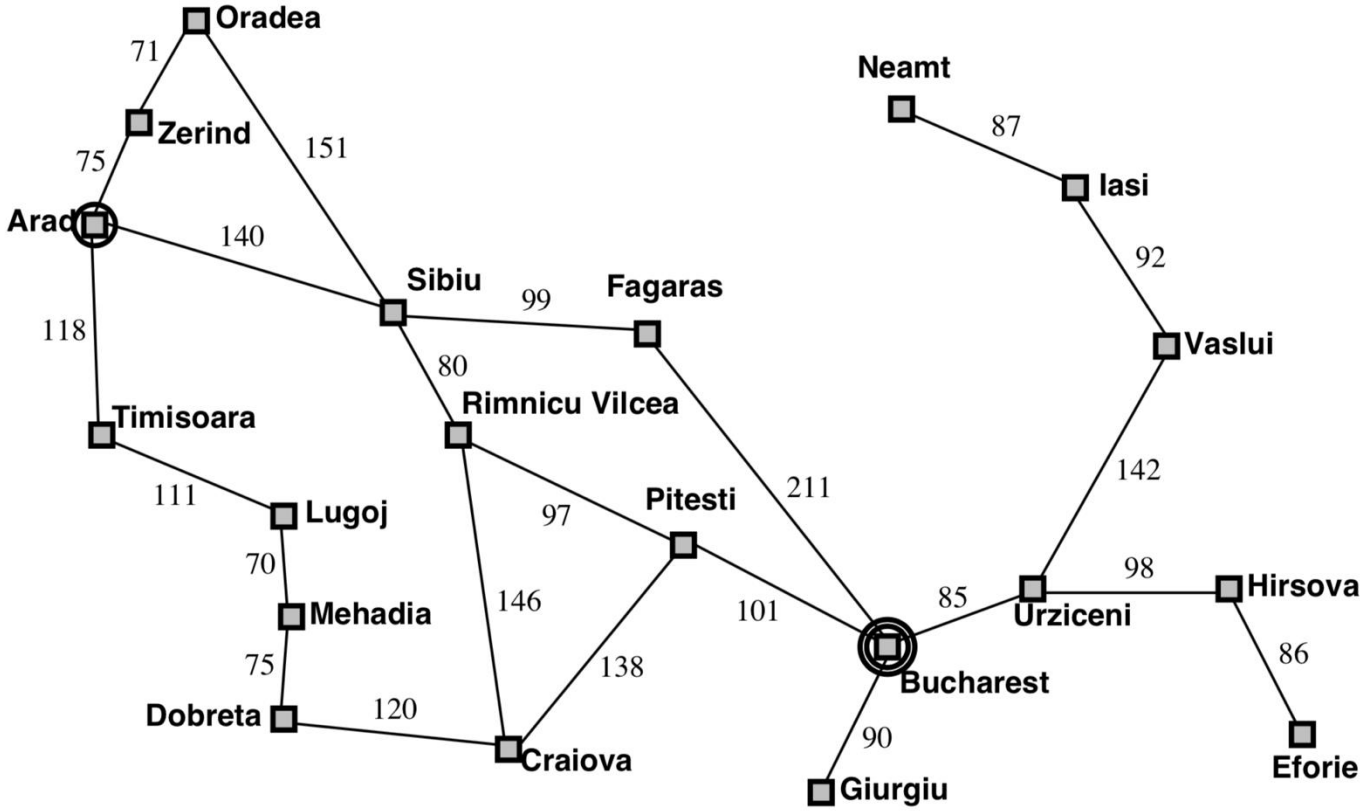
3. **Execute** this behavior
   - The bottomline, drives all of the above





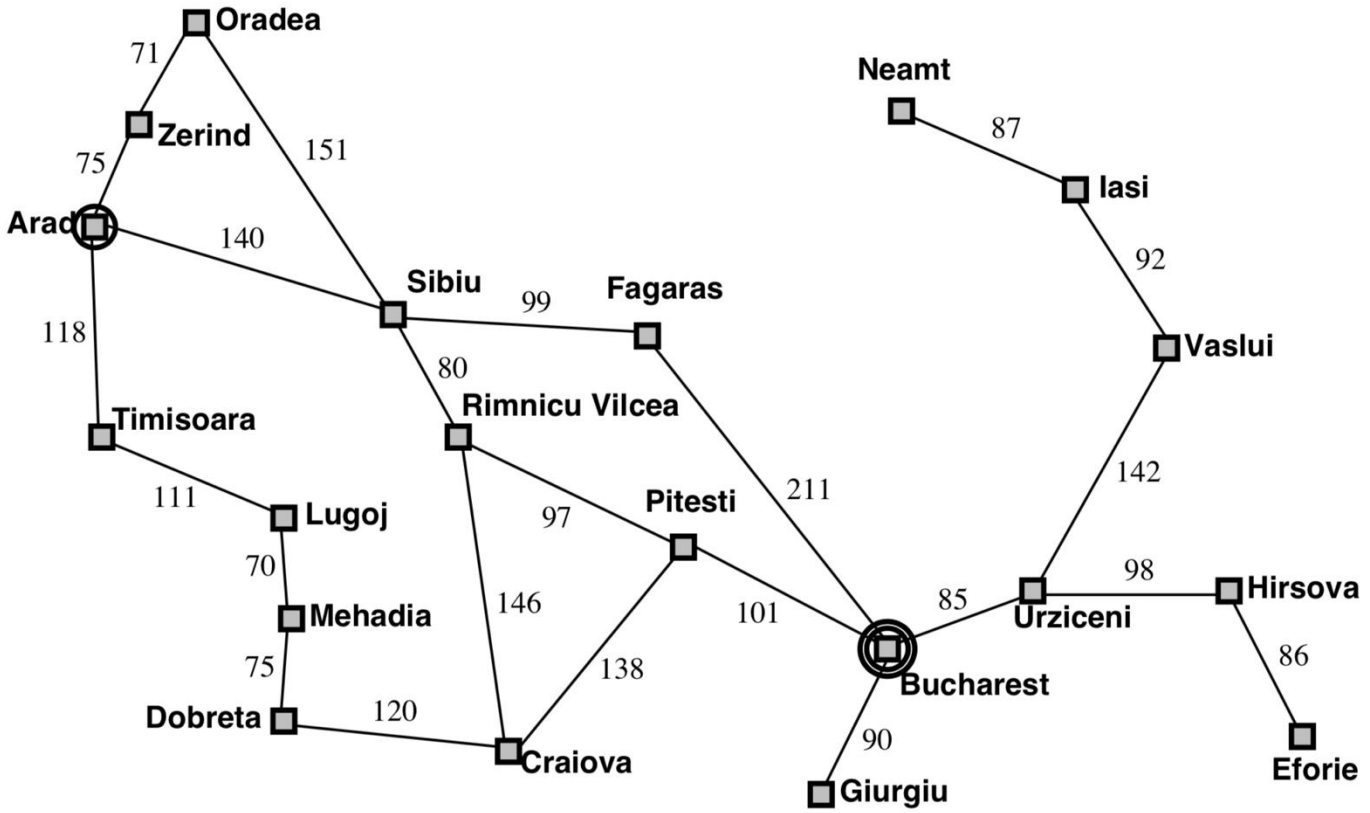
Route-Planning Agent

- State:
  - <current city>
- Action:
  - Move along edge
- Performance measure:
  - Cost (length) of path from initial city to goal city



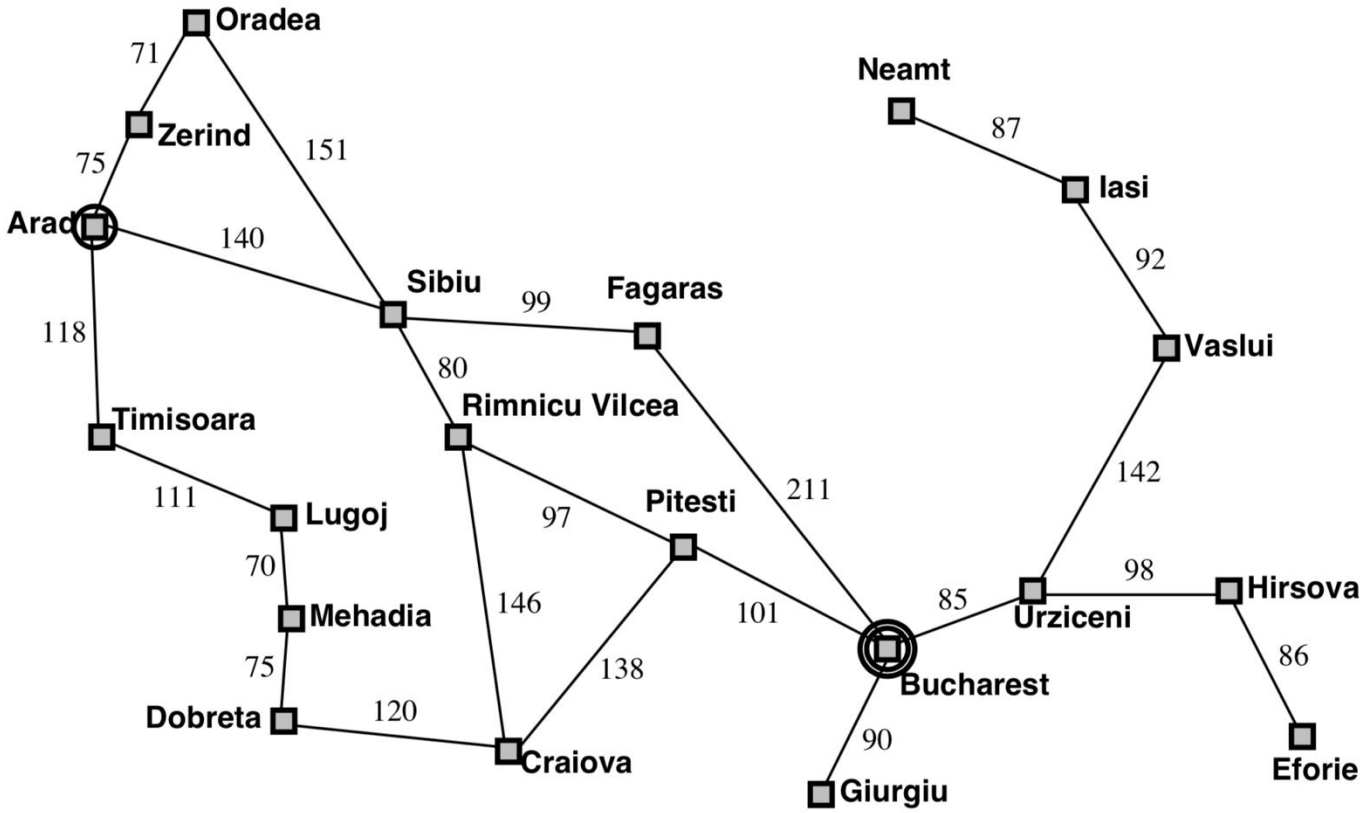
Route-Planning Agent

- State:
  - <current city>
- Action:
  - Move along edge
- Performance measure:
  - Cost (length) of path from initial city to goal city
- Such problems are known as *search problems*



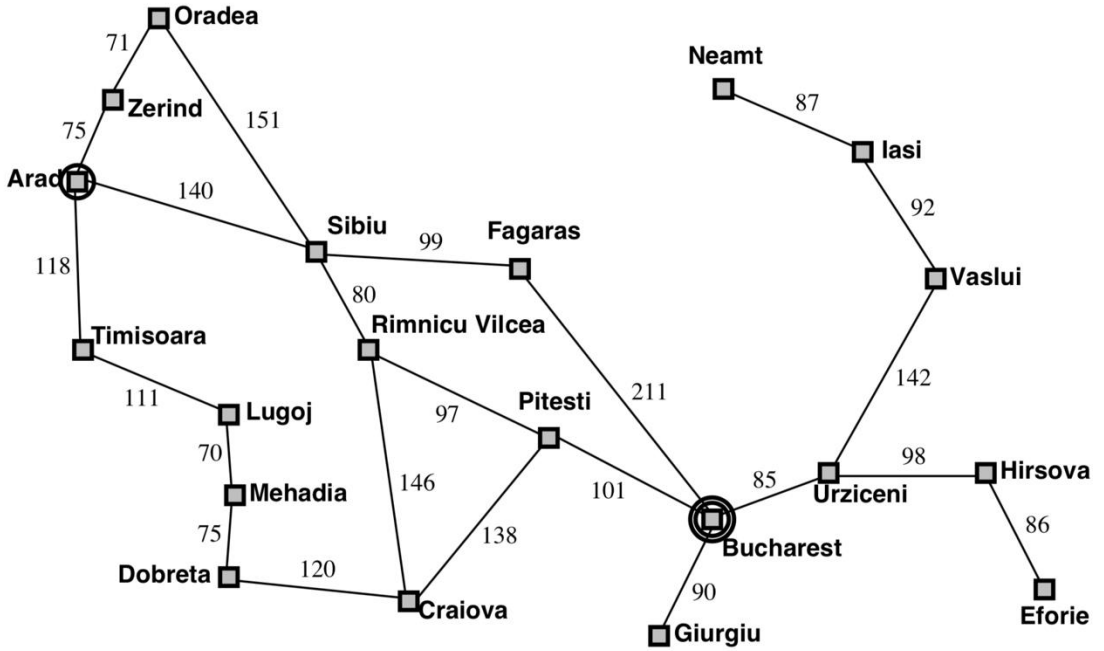
Route-Planning Agent

- Examples of search algorithms?



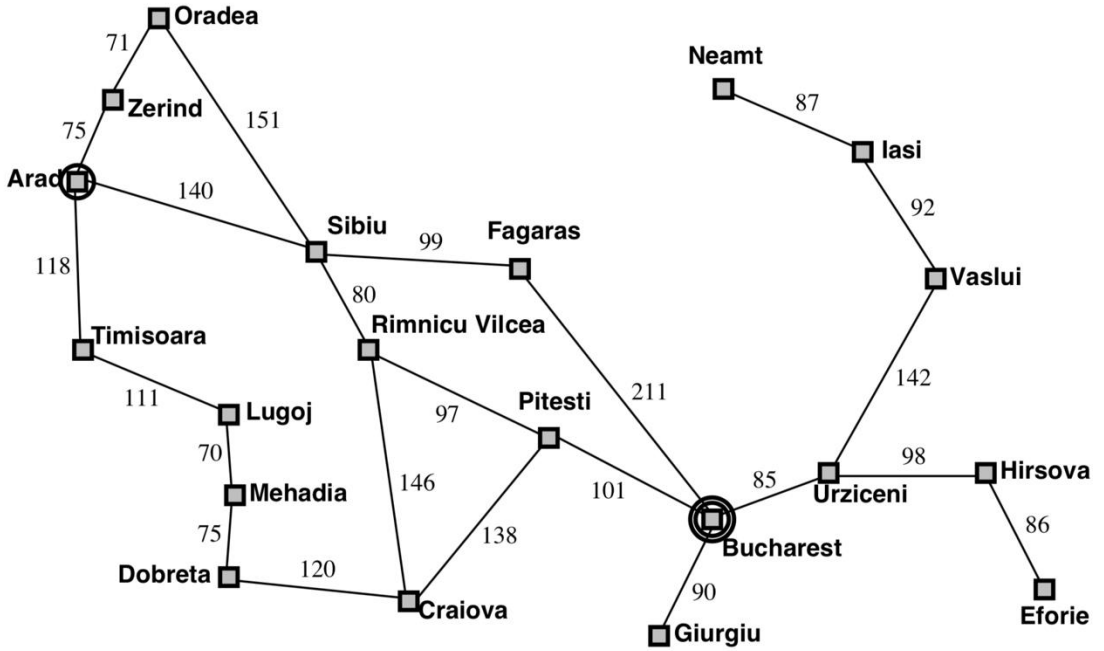
Evaluating Search Algorithms

- Properties of algorithms for computing paths (agent programs):
  - Completeness
    - The algorithm will **find a solution if there exists one**
  - Soundness
    - If the algorithm returns a solution, it will be a valid path in the graph
  - Space complexity
  - Time complexity
  - Optimality
    - Algorithm returns solutions of *minimal* cost



Evaluating Search Algorithms

- Search problems go beyond route planning
- All that's needed:
  - Deterministic actions
  - Fully observable state
  - Initial state, goal state(s)
- What types of problems can be expressed as a graph?

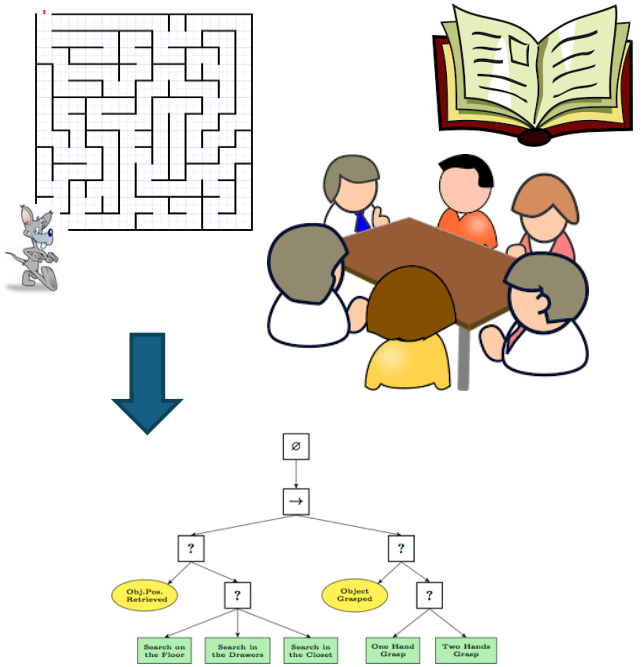


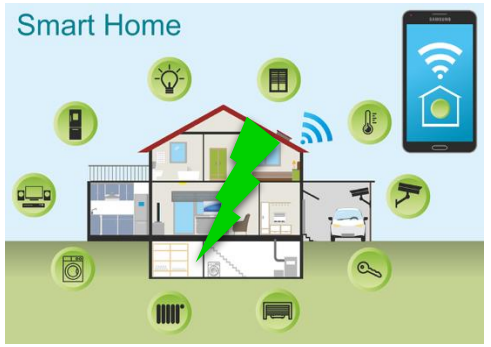
The Typical Agent Design Process

1. **Represent** background information and objective
   - Multi-step agents require states, actions and their effects on states
   - Learned using simulator, modeled using literature/experts, or partially modeled with scope for online learning

2. **Compute** "behavior" that would allow agent to achieve objective
   - Behavior may include contingencies depending on observed system state during execution

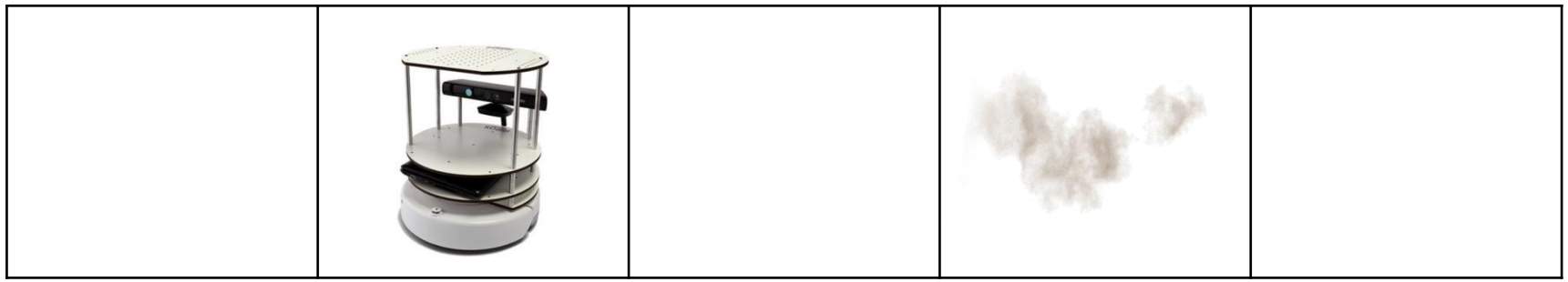
3. **Execute** this behavior
   - The bottomline, drives all of the above





Problem Types and Solution Representations

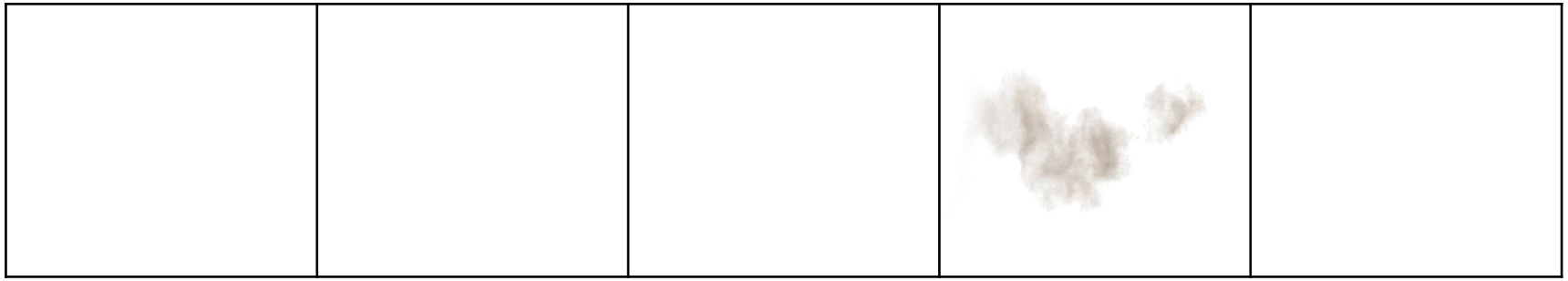
- Deterministic, fully observable problem
  - $T: S \times A \rightarrow S$; agent observes full state
  - "Classical planning": solution is a plan, or a sequence of actions



Solution plan: right, right, suck

Forms of Solutions

- Deterministic, fully observable
  - "Classical planning": solution is a plan, or a sequence of actions
- Deterministic, non-observable
  - "Conformant planning": solution is a plan, but often no solutions exist



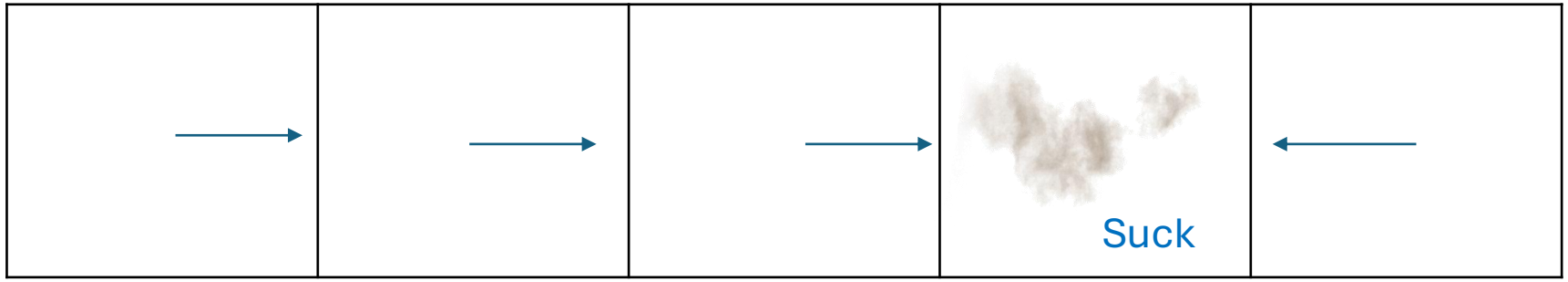
???

Right, Right, Right, Right, Suck, Left, Suck, Left, Suck, Left, Suck, Left, Suck

Forms of Solutions

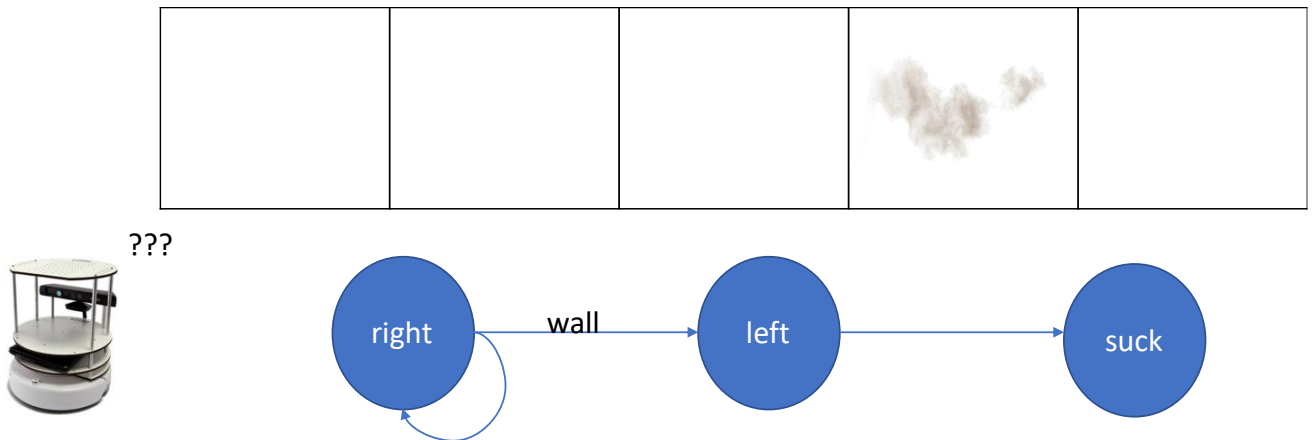
- Deterministic, fully observable
  - "Classical planning": solution is a plan, or a sequence of actions
- Deterministic, non-observable
  - "Conformant planning": solution is a plan, but often no solutions exist
- Non-deterministic, fully observable:
  - "Contingent planning": solution can be a policy (a function $S \rightarrow A$)

Non-determinism: a move action may fail



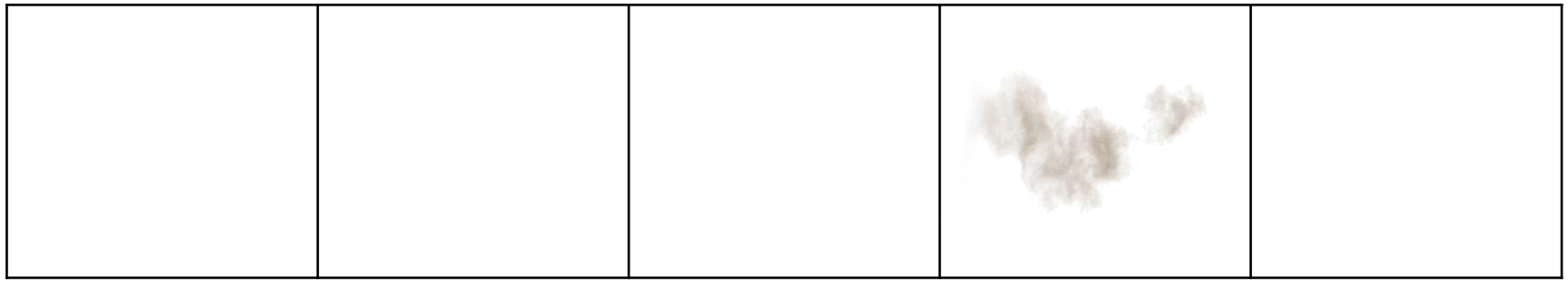
Forms of Solutions

- Deterministic, fully observable
  - "Classical planning": solution is a plan, or a sequence of actions
- Deterministic, non-observable
  - "Conformant planning": solution is a plan, but often no solutions exist
- Non-deterministic, fully observable:
  - "Contingent planning": solution can be a policy (a function $S \rightarrow A$)
- Partially observable
  - Solution is a finite state machine

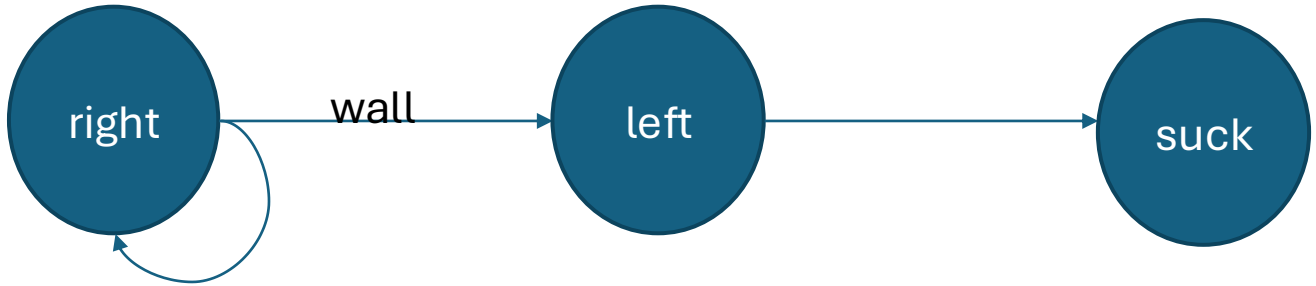


Forms of Solutions

- Deterministic, fully observable
  - "Classical planning": solution is a plan, or a sequence of actions
- Deterministic, non-observable
  - "Conformant planning": solution is a plan, but often no solutions exist
- Non-deterministic, fully observable:
  - "Contingent planning": solution can be a policy (a function $S \rightarrow A$)

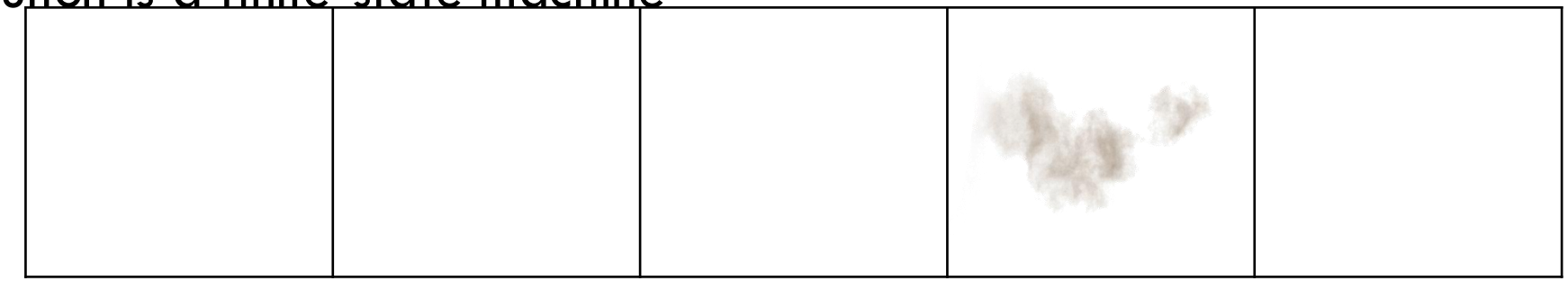


???

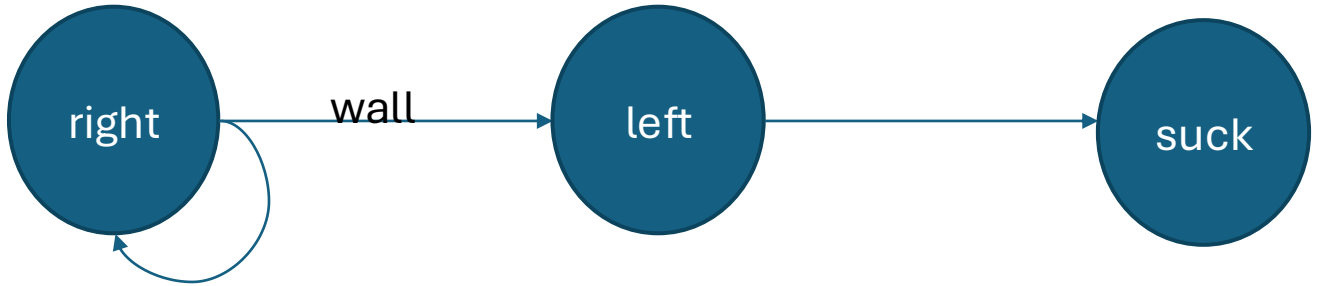


Forms of Solutions

- Deterministic, fully observable
  - "Classical planning": solution is a plan, or a sequence of actions
- Non-observable
  - "Conformant planning": solution is a plan, but often no solutions exist
- Partially observable
  - Solution is a finite-state machine



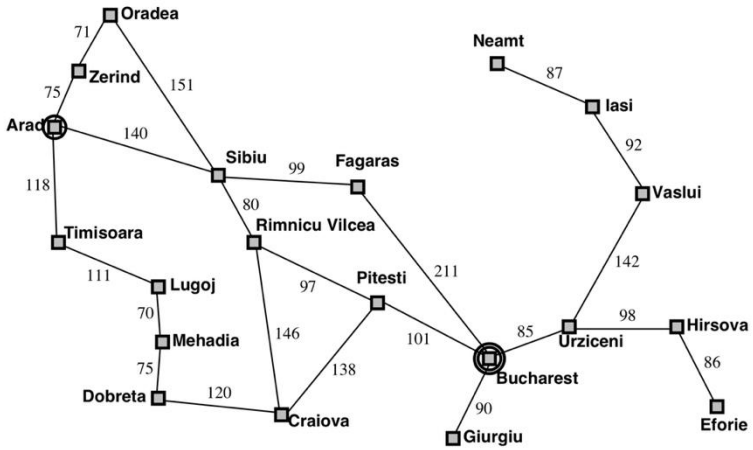
???



Deep Dive: Deterministic and Fully Observable Systems

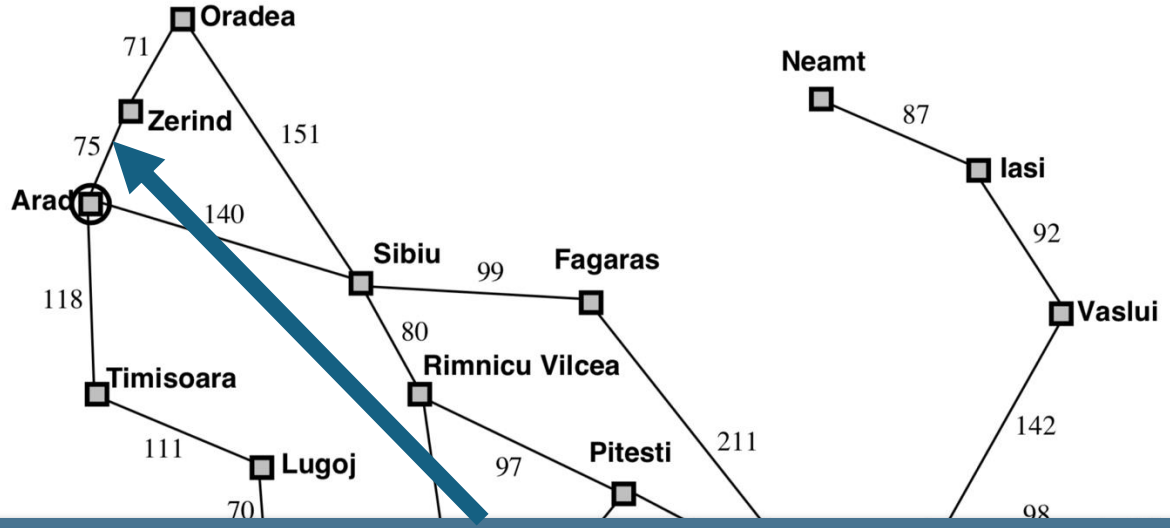
Naïve Formulation of Search Problems

- Given:
  - Graph $\langle V, E, A, \ell \rangle$
    - V set of vertices
    - E set of directed edges
    - A set of actions
    - $\ell: E \rightarrow \langle A, \mathbb{R}^+ \rangle$
  - $v_i \in V$ start vertex
  - G goal vertex set
- Find a path from v_i to G



Naïve Formulation of Search Problems

- Given:
  - Graph $\langle V, E, A, \ell \rangle$
    - V set of vertices
    - E set of directed edges
    - A set of actions
    - $\ell: E \rightarrow \langle A, \mathbb{R}^+ \rangle$
  - $v_i \in V$ start vertex
  - G goal vertex set
- Find a path from v_i
- The directed edge f



(Arad, Zerind)
(Zerind, Arad)

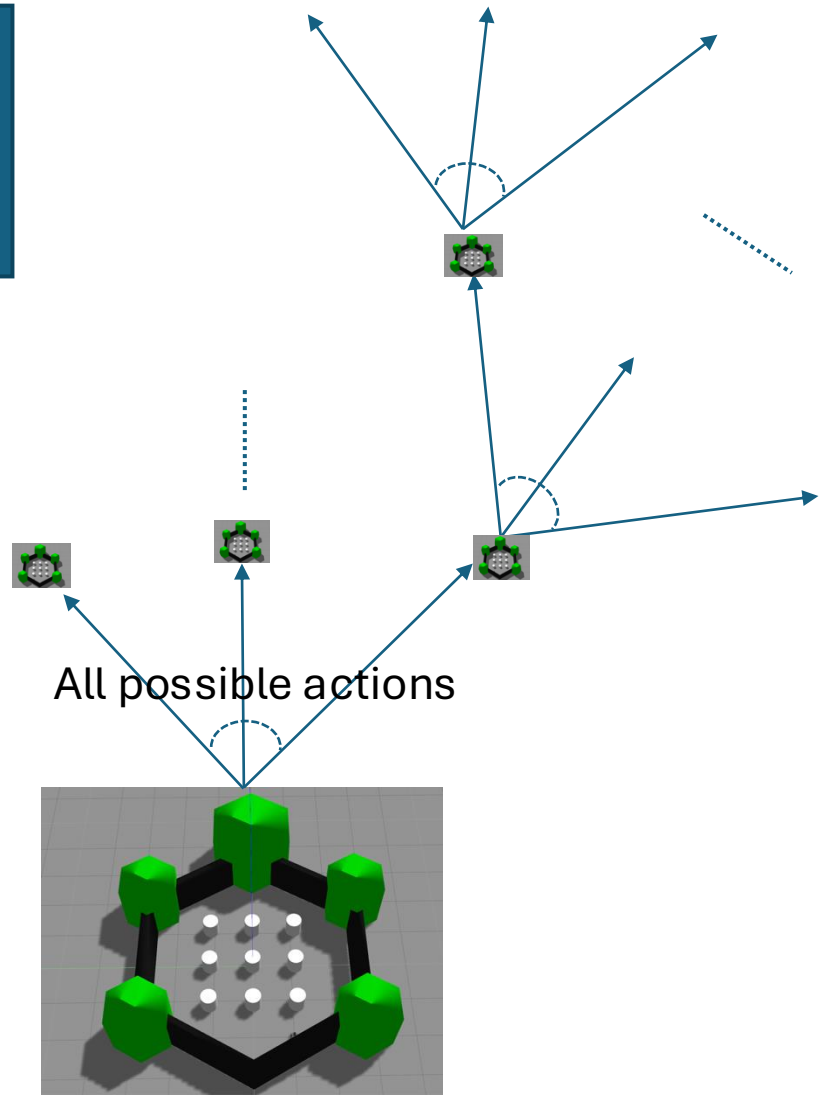
$\ell\big((Arad, Zerind)\big) = \langle drive\_Arad\_Zerind, 75 \rangle$

$\ell\big((Zerind, Arad)\big) = \langle drive\_Zerind\_Arad, 75 \rangle$

Problem with Naïve Formulation of Search Problems

- Given:
  - Graph $\langle V, E, A, \ell \rangle$
    - V set of vertices
    - E set of directed edges
    - A set of actions
    - $\ell: E \rightarrow \langle A, \mathbb{R}^+ \rangle$
  - $v_i \in V$ start vertex
  - G goal vertex set
- Find a path from v_i to G

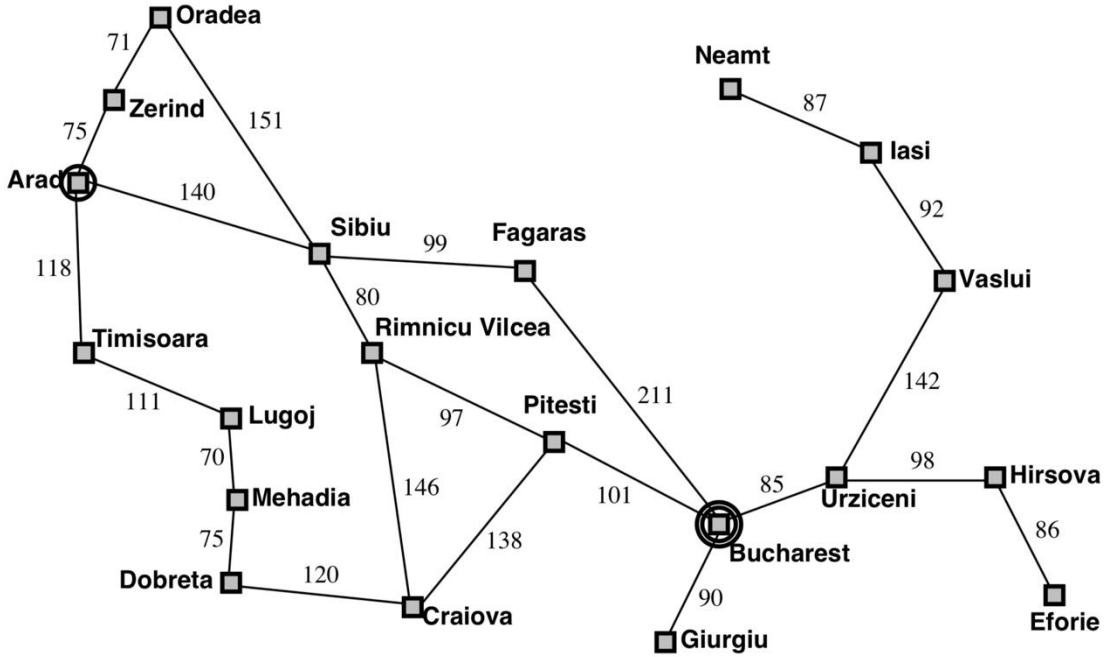
In most cases, the full graph is too large to specify explicitly



Search Problem Defined Over an Implicit Graph

- A search problem is defined by
  - S set of possible states
    - Defined, but not enumerated a priori
  - $s_0 \in S$ start state (~vertex)
  - A set of actions
  - $N: S \rightarrow S$ successor function
    - $T: S \times A \rightarrow S$ deterministic transition function
    - $N(s) = \{s' : \exists a \, T(s, a) = s'\}$

Set of all s' that can be reached in one step from s

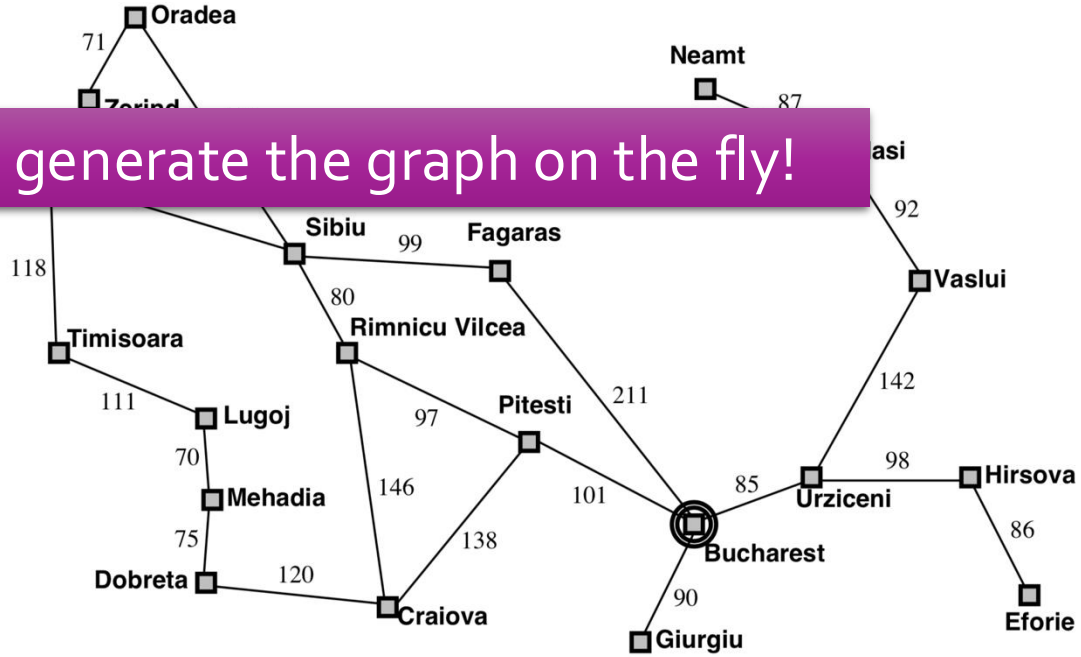


- $N(Arad) = \{Timisoara, Sibiu, Zerind\}$

Search Problem Defined Over an Implicit Graph

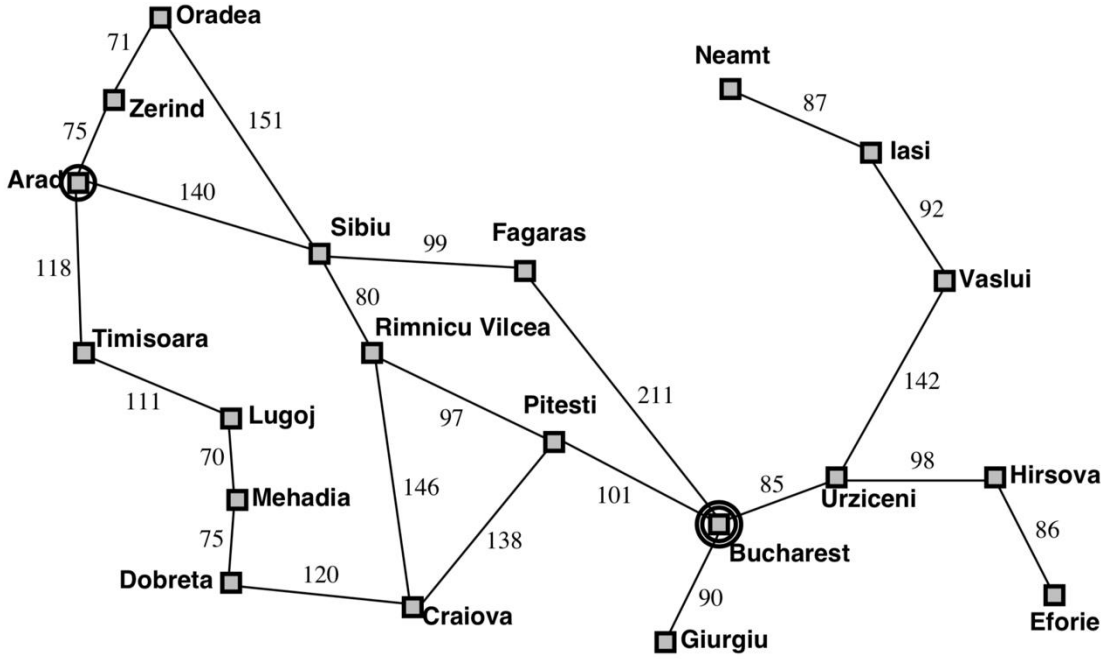
- A search problem is defined by
  - S set of possible states
    - Defined, but not enumerated a priori
  - $s_0 \in S$ start state (~vertex)
  - A set of actions
  - $N: S \to S$ successor function
    - $T: S \times A \to S$ deterministic transition function
    - $N(s) = \{s' : \exists a\, T(s,a) = s'\}$

Allows us to generate the graph on the fly!



Search Problem Defined Over an Implicit Graph

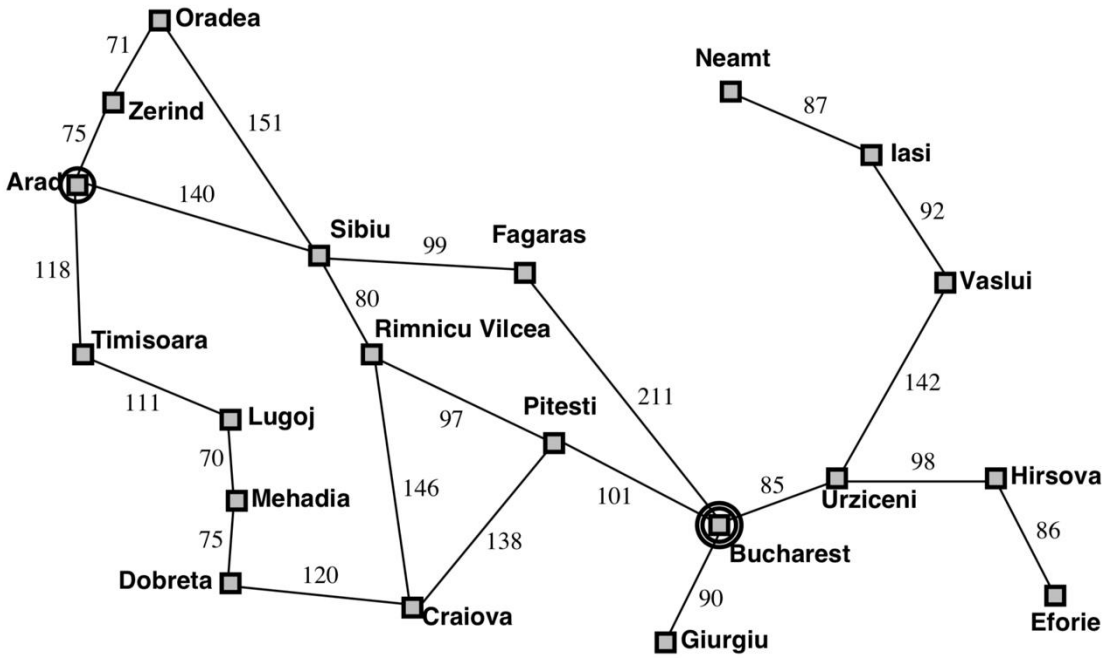
- A search problem is defined by
  - S set of possible states
    - Defined, but not enumerated a priori
  - $s_0 \in S$ start state (~vertex)
  - A set of actions
  - $N: S \rightarrow S$ successor function
    - $T: S \times A \rightarrow S$ deterministic transition function
    - $N(s) = \{s' : \exists a \, T(s, a) = s'\}$
  - $G: S \rightarrow \{True, False\}$ goal test
  - $C: S \times A \times S \rightarrow \mathbb{R}^+$ path cost



A **solution** to a search problem is a sequence of actions leading from s_o to a state satisfying the goal test

Role of Abstraction in Modeling Search Problems

- Search problem formulations are models
  - Most models are abstractions
  - Driving from one city to another involves
    - Traffic, one-way streets, fuel considerations, rest stops, …
- Abstract state = set of real states
- Abstract action = set of procedures

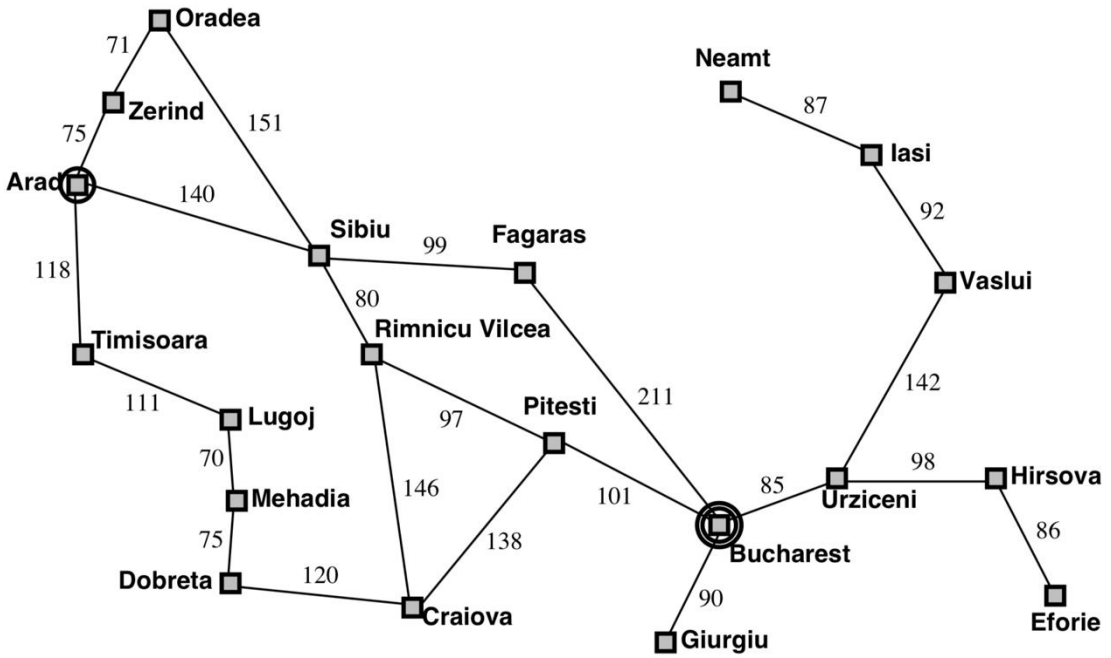


Abstraction is a common and important aspect of modeling

Abstract problem should be easier than the real problem!

Role of Abstraction in Modeling Search Problems

- For the abstract solution to be usable we need:
  - $\forall s, a, s'$ such that $T(s, a) = s'$
    - Every real state in s gets to some real state in s' by following a procedure defined by a
    - "Downward refinability"
  - There should be a way to go from all real locations "in Arad" to some real location "in Zerind"

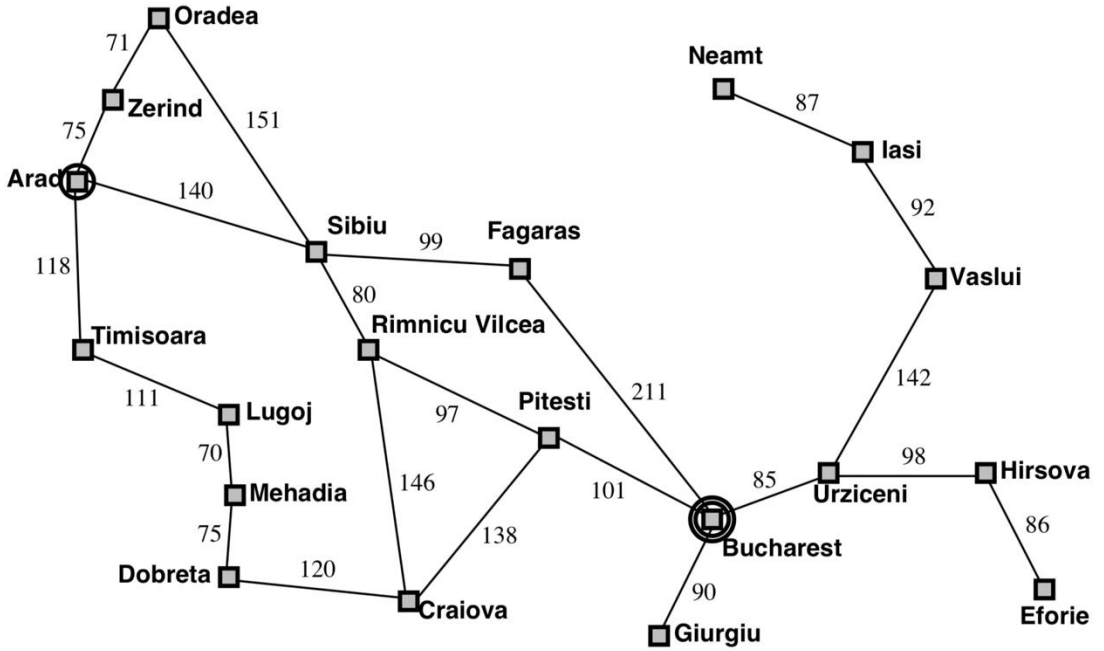


Abstract problem should be easier than the real problem!

Solving Search Problems

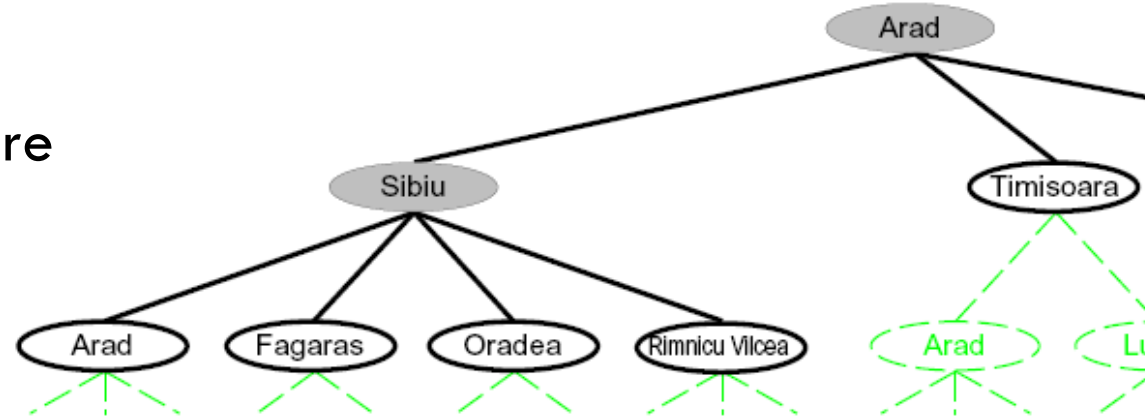
Evaluating Search Algorithms

- Properties of algorithms for computing paths (agent programs):
  - Completeness
    - The algorithm will find a solution if there exists one
  - Soundness
    - If the algorithm returns a solution, it will be a valid path in the graph
  - Space complexity
  - Time complexity
  - Optimality
    - Algorithm returns solutions of *minimal* cost



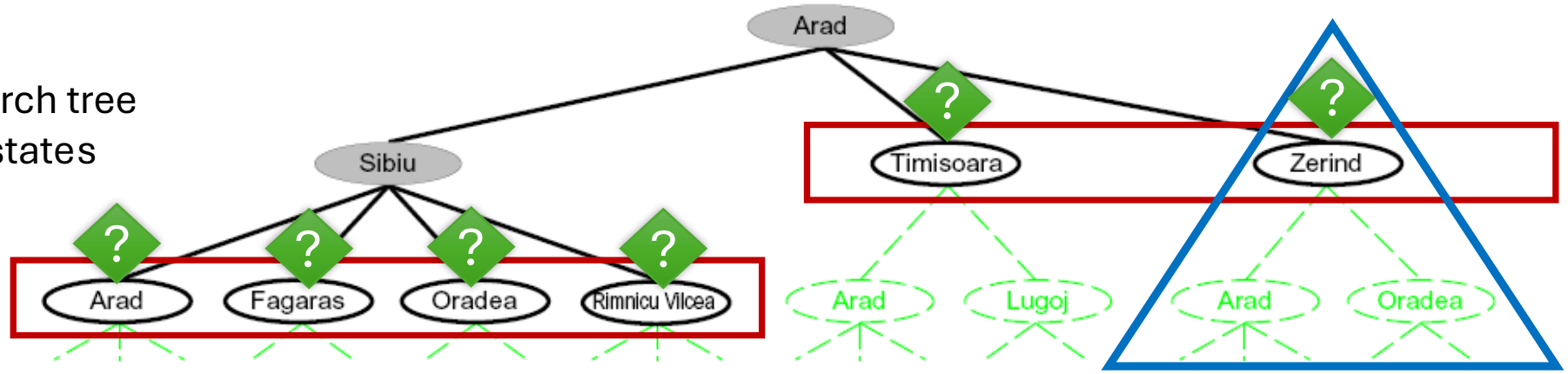
Search Algorithms: Fundamentals

- A search tree is used to evaluate possible courses of action in an offline simulation
  - Use the successor function to incrementally explore neighbors of explored states
  - ➢ "Expand" states
- Node in the search tree vs state
  - A state is a configuration of the agent, environment
  - A node is a data structure used in the search-tree
    - State, parent, children, depth, incurred path cost $g(x)$
    - Multiple nodes may include the same state (not desirable)
    - Each node corresponds to a partial plan
  - The root node includes the start state



General Notions of Search

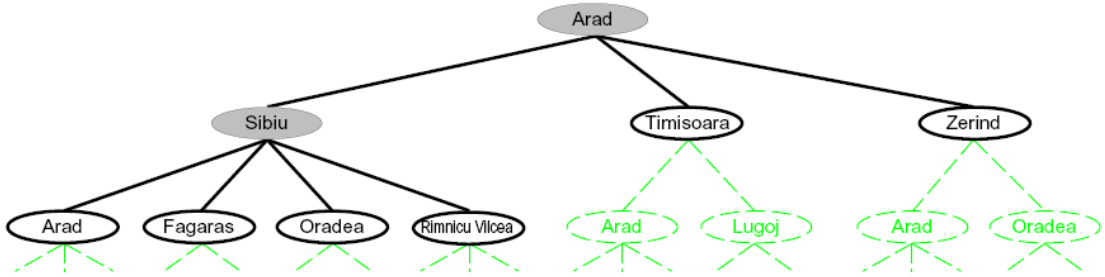
Partial search tree
Nodes vs states



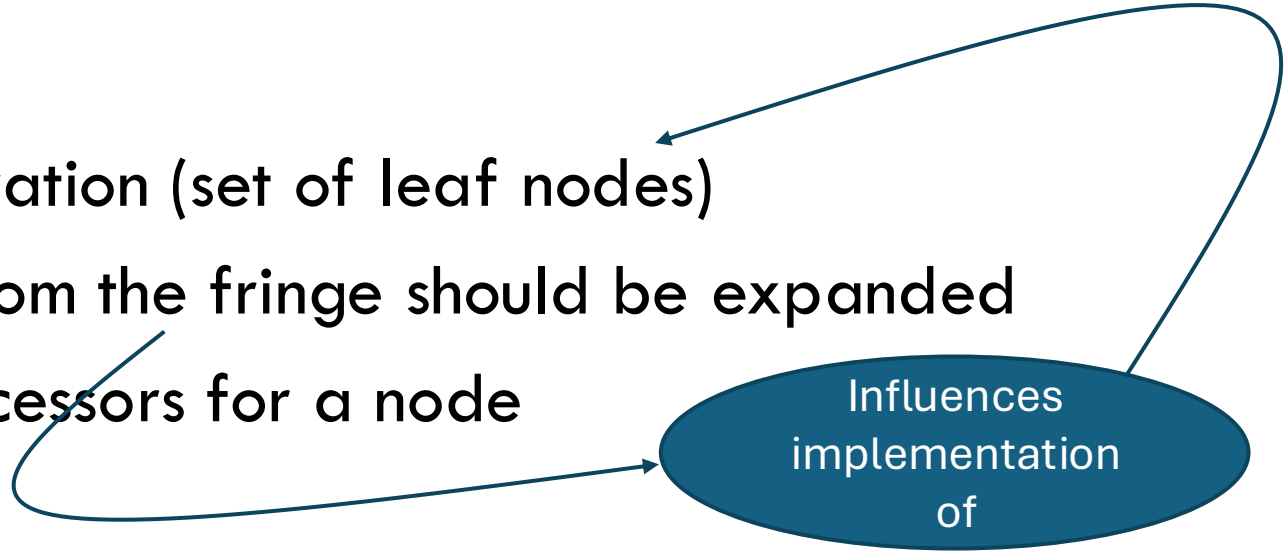
- Fringe: partial plans under consideration
- Exploration strategy: which node from the fringe should be expanded
- Node expansion: generation of successors for a fringe node

General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

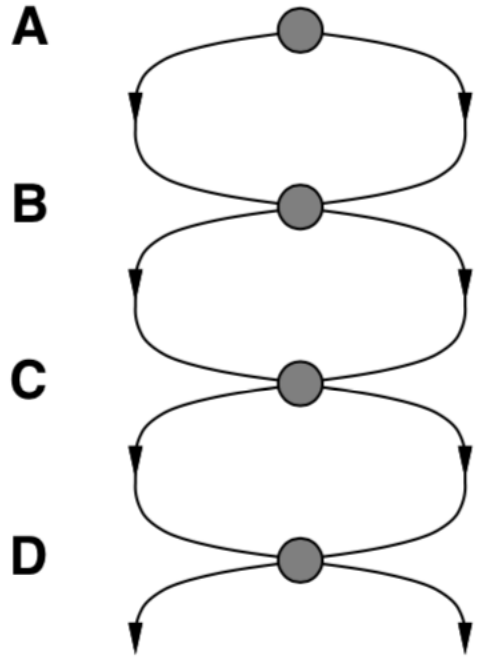


- Fringe: partial plans under consideration (set of leaf nodes)
- Exploration strategy: which node from the fringe should be expanded
- Node expansion: generation of successors for a node



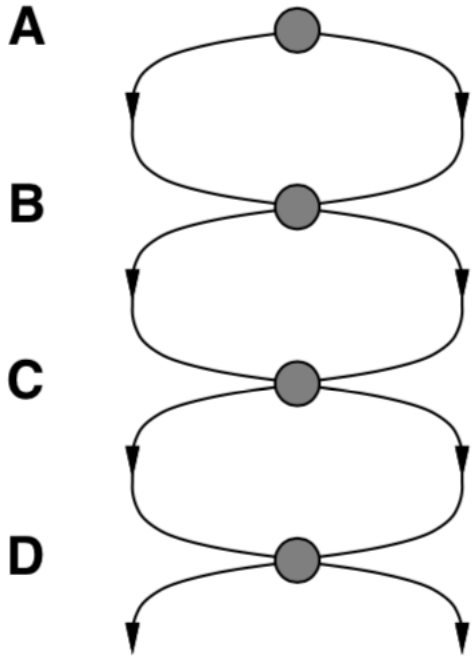
Problem with Tree Search

Doesn't detect repeated states



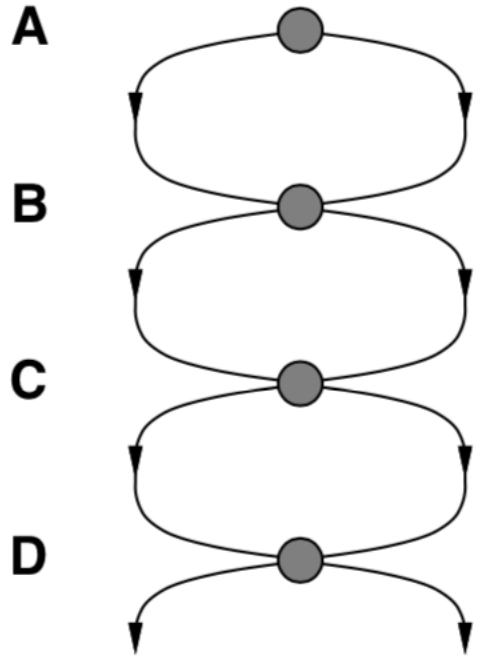
Problem with Tree Search

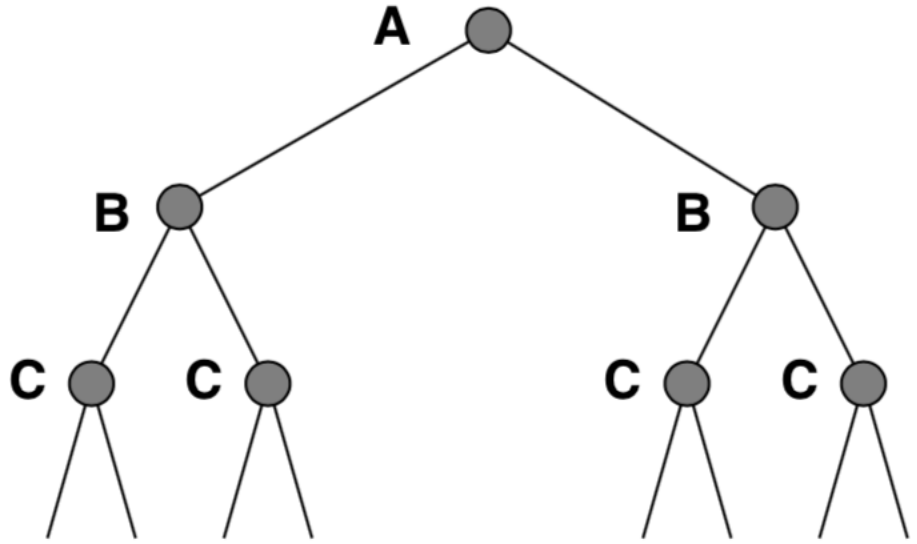
Doesn't detect repeated states



Problem with Tree Search

Doesn't detect repeated states





Easy Fix: Maintain a *Closed* List

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Easy Fix: Maintain a Closed List

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

For clarity in illustrations, we will use the tree-search variant

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Uninformed Search Algorithms

Uniformed Search

- Uninformed, or undirected search algorithms operate without any consideration of how far a state is from a goal
  - Good for situations where distance to a goal cannot be estimated

- We will learn about:
  - Depth first search
  - Breadth first search
  - Iterative deepening search
  - Uniform cost search

Depth-First Search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

- Exploration strategy (Remove-Front): returns the deepest node
- Fringe is implemented as a LIFO stack

Videos of some these algorithms: https://www.movingai.com/SAS/EXP/

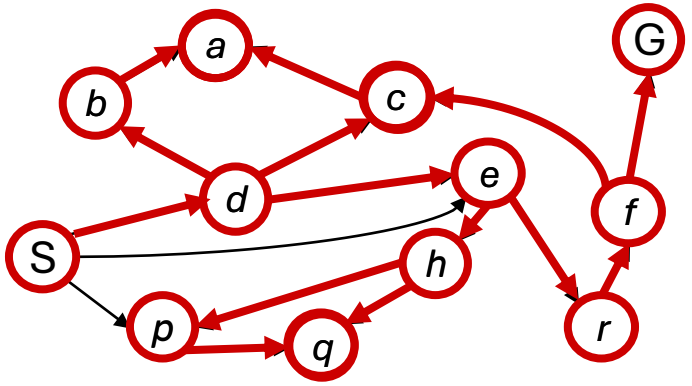
Organized by Nathan Sturtevant, University of Denver

Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack

(successors go on top)



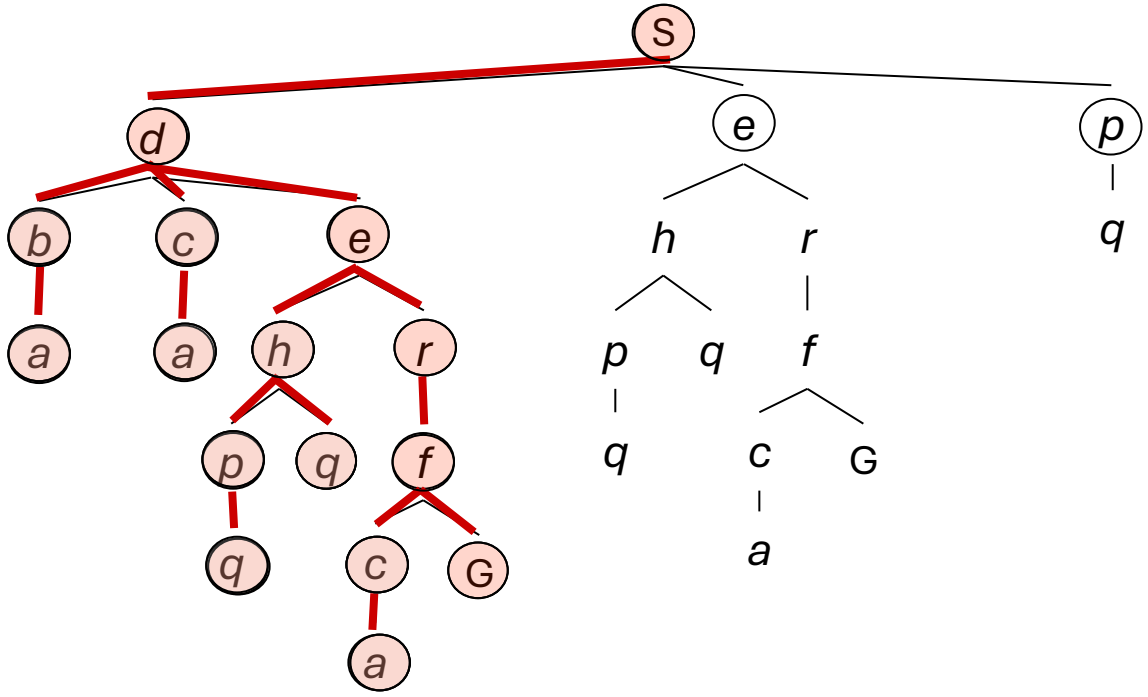


Illustration courtesy Abeel and Kleir

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots. b^m = O(b^m)$

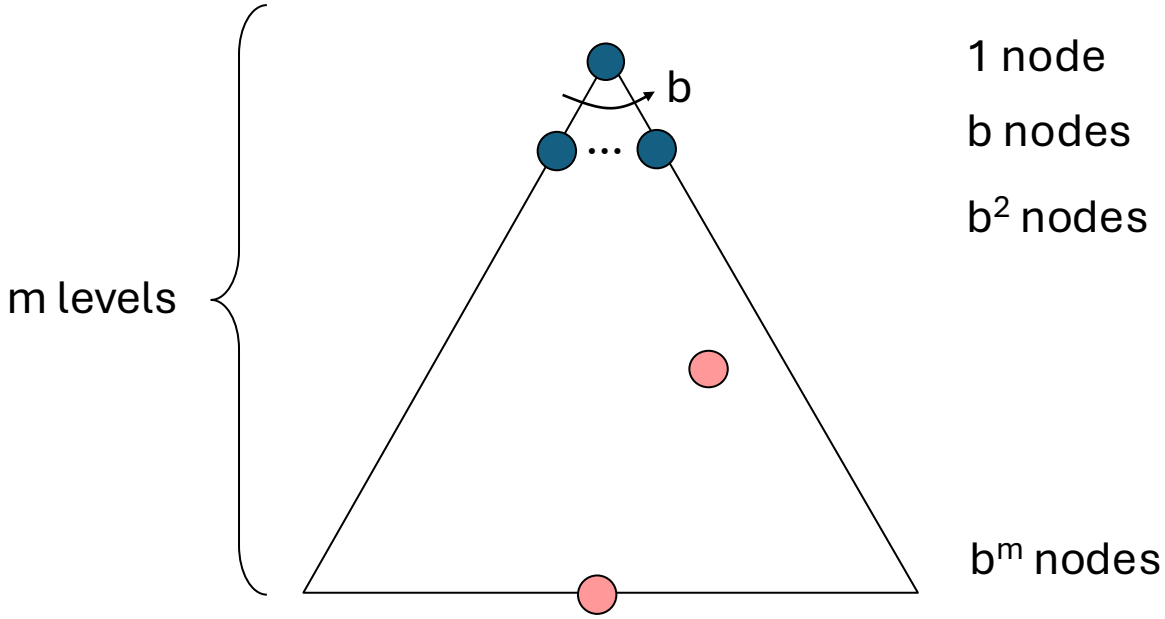


Illustration courtesy Abeel and Kleir

Depth-First Search (DFS) Properties

- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so O(bm)
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost

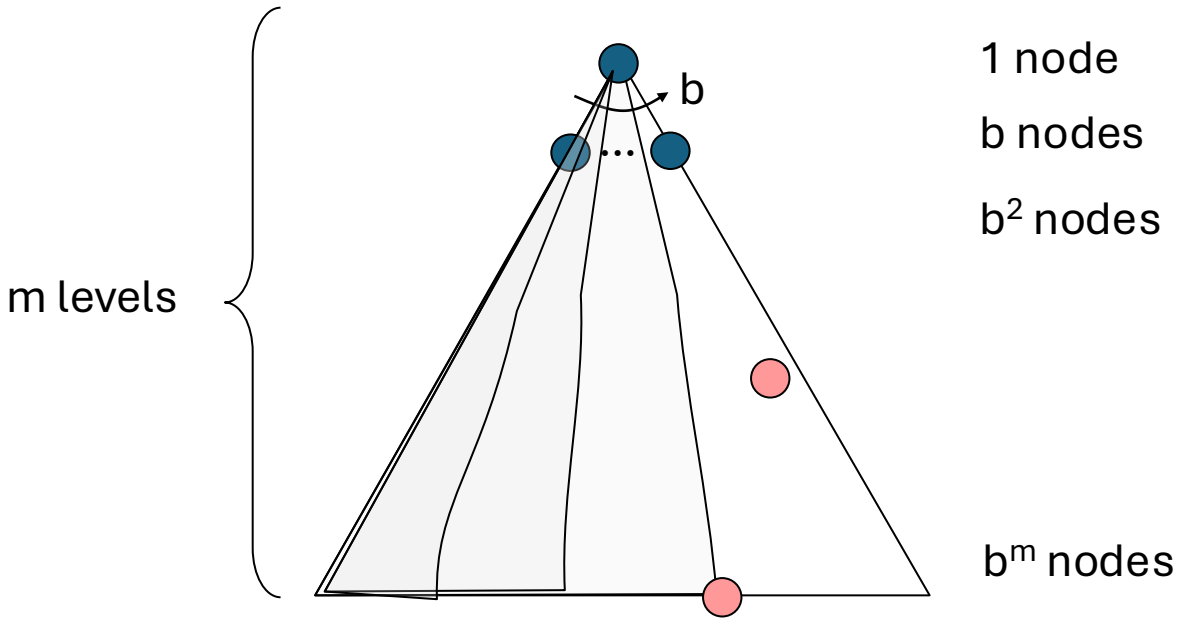


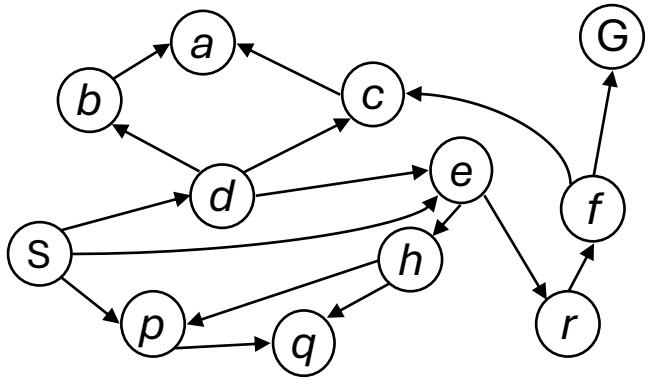
Illustration courtesy Abeel and Kleir

Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

(successors go at end)



Search Tiers

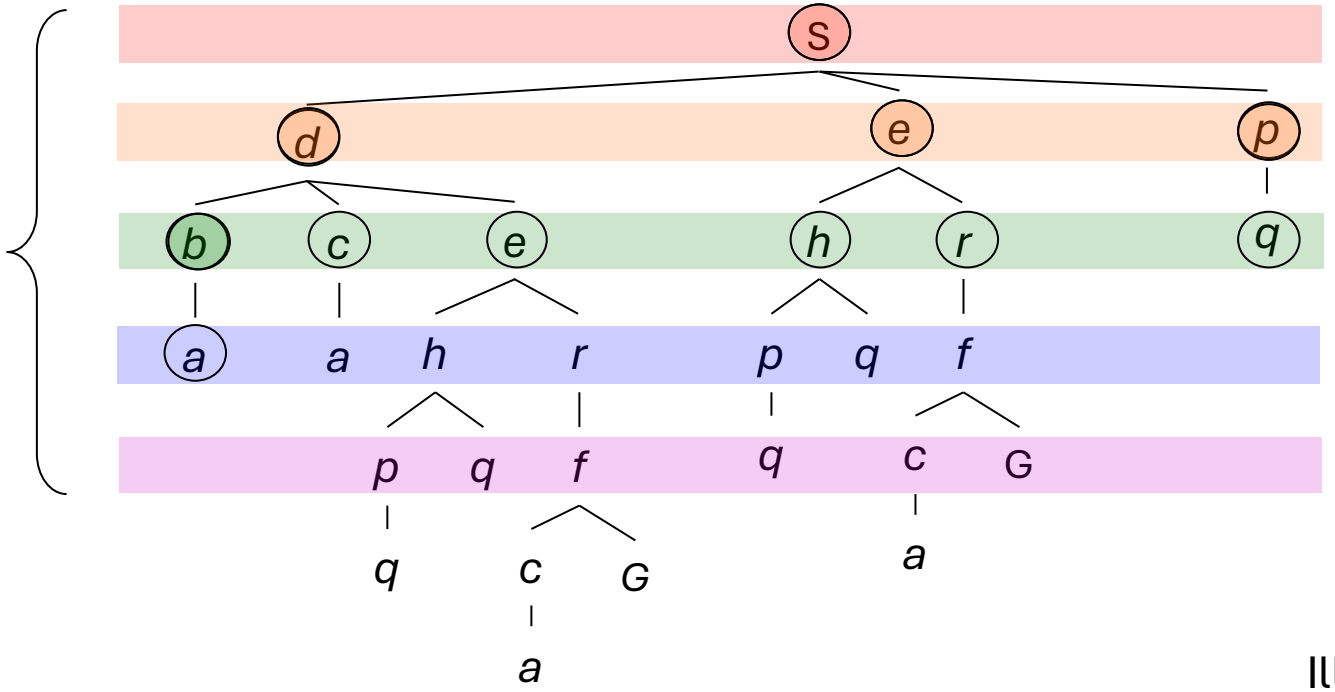


Illustration courtesy Abeel and Kleir

Analysis of BFS

- (See the textbook)
- Time complexity: $O(b^s)$, where s is depth of shallowest goal node
- Space complexity: $O(b^s)$
- Complete?
- Optimal?

Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1 (more on costs later)

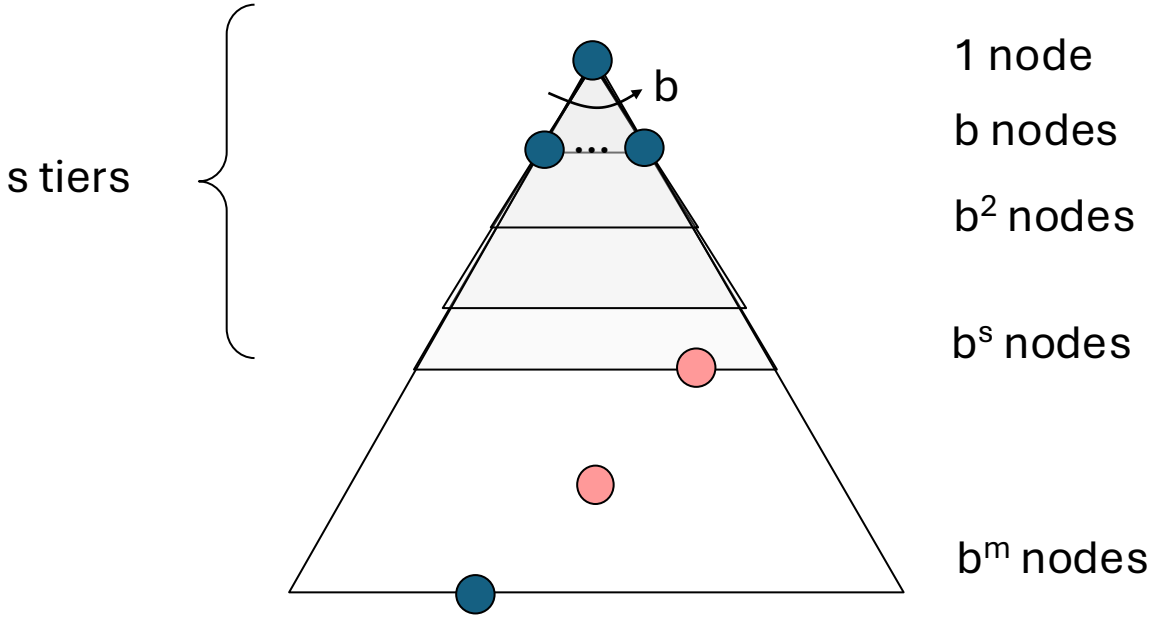


Illustration courtesy Abeel and Kleir

- Which would be better for a computational-space constrained scenario?

- Suppose solution is at depth s << m; b>2. What would you use?

Iterative Deepening

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run DFS with depth limit 1. If no solution…
  - Run DFS with depth limit 2. If no solution…
  - Run DFS with depth limit 3. …..
- Why is this a good idea?
- BFS generates successor nodes at the next level (can be fixed: how?)
- Space usage $O(bs)$ vs $O(b^s)$

$$b = 10, m = 5$$

$$N(IDS) = 5 \cdot 10 + 4 \cdot 100 + 3 \cdot 10^3 + 2 \cdot 10^4 + 10^5$$

$$N(BFS) = 10 + 100 + 10^3 + 10^4 + 10^5 + 999{,}990$$

$$(s+1) \cdot 1 + s \cdot b + (s-1) \cdot b^2 + \cdots + 1 \cdot b^s \sim O(b^s)$$

Cost-Sensitive Search

- BFS expands the graph “uniformly” in terms of depth
- All nodes at a lower depth are expanded before nodes at a greater depth
- Often, actions (edges) have heterogeneous costs, so expanding lower depths may yield expensive solutions
- Can BFS be extended to expand lower-cost nodes first?

Yes!