CS 3630!

Lecture 2:

Sense, Think, Act

Sense, Think, Act

Suppose you are given a task: *Rearrange the chairs in the room into a circle.* How would you proceed?

1. Look around the room and evaluate the situation.
   Where are the chairs? How many chairs are there?

Sense

2. Make a plan:
   1. Go the first chair, pick it up, place it in the desired position
   2. Repeat for all N chairs.

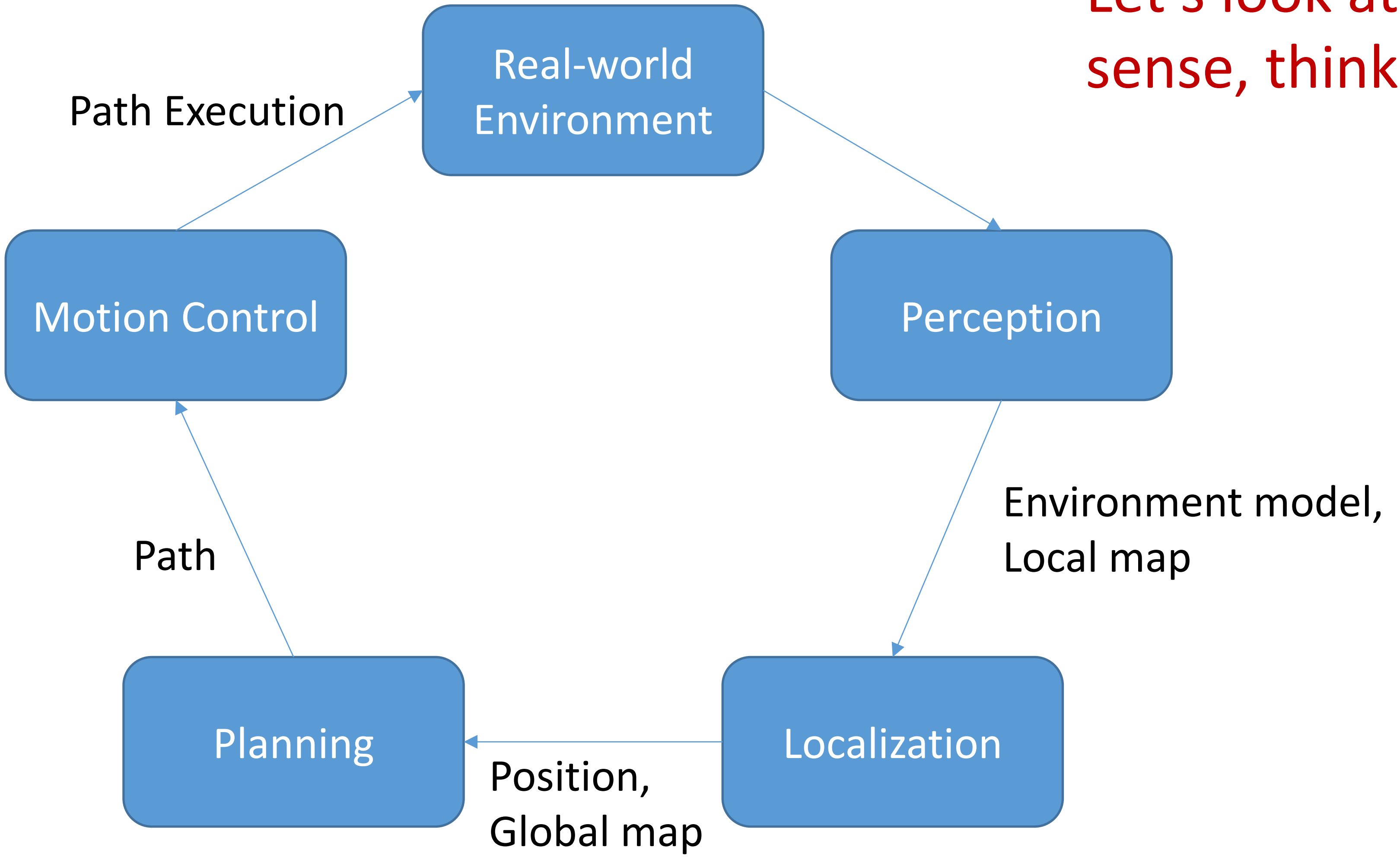
Think

3. Execute the plan.

Act

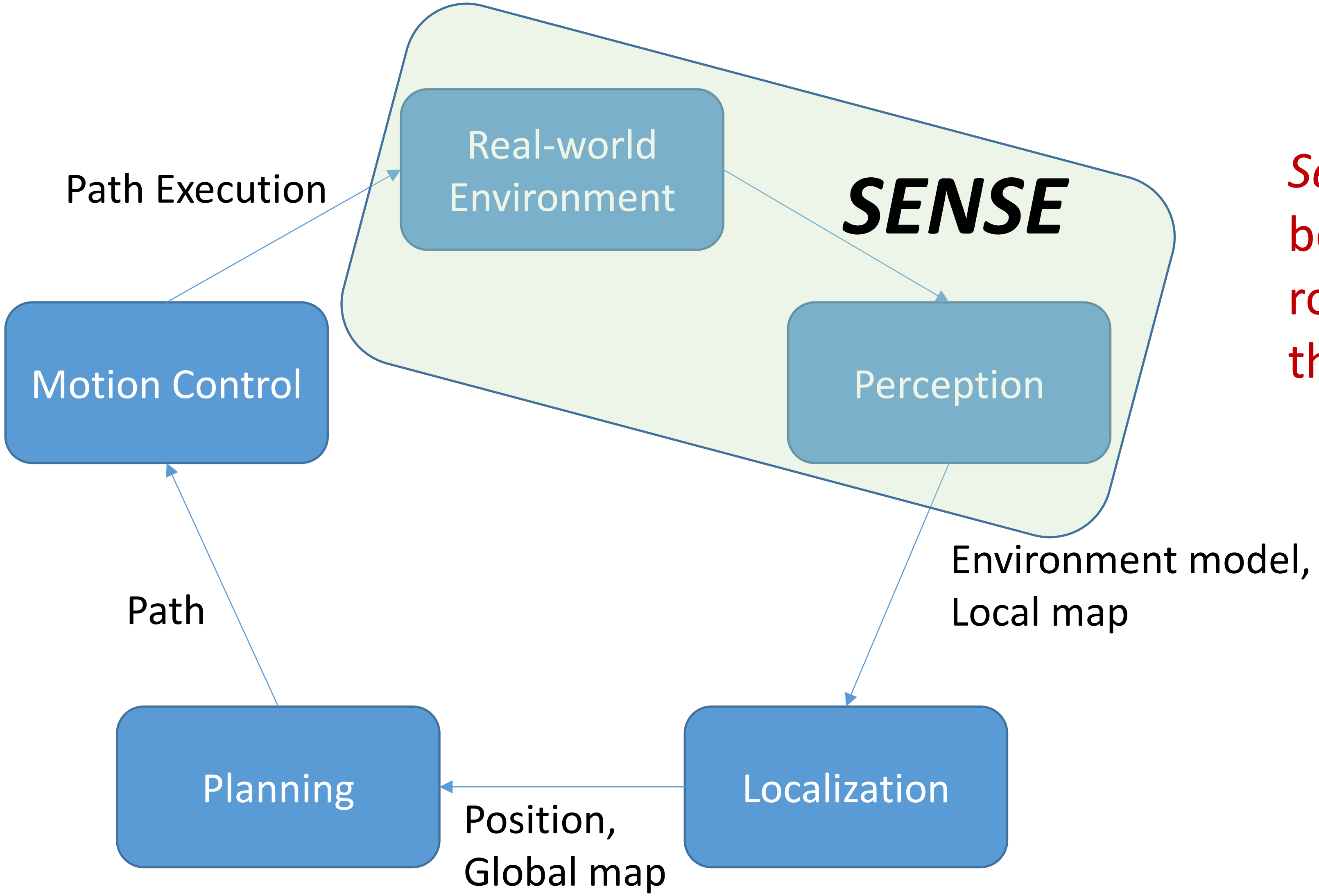
This is the basic strategy followed by almost all robots.

Example: Navigation in a Known Environment



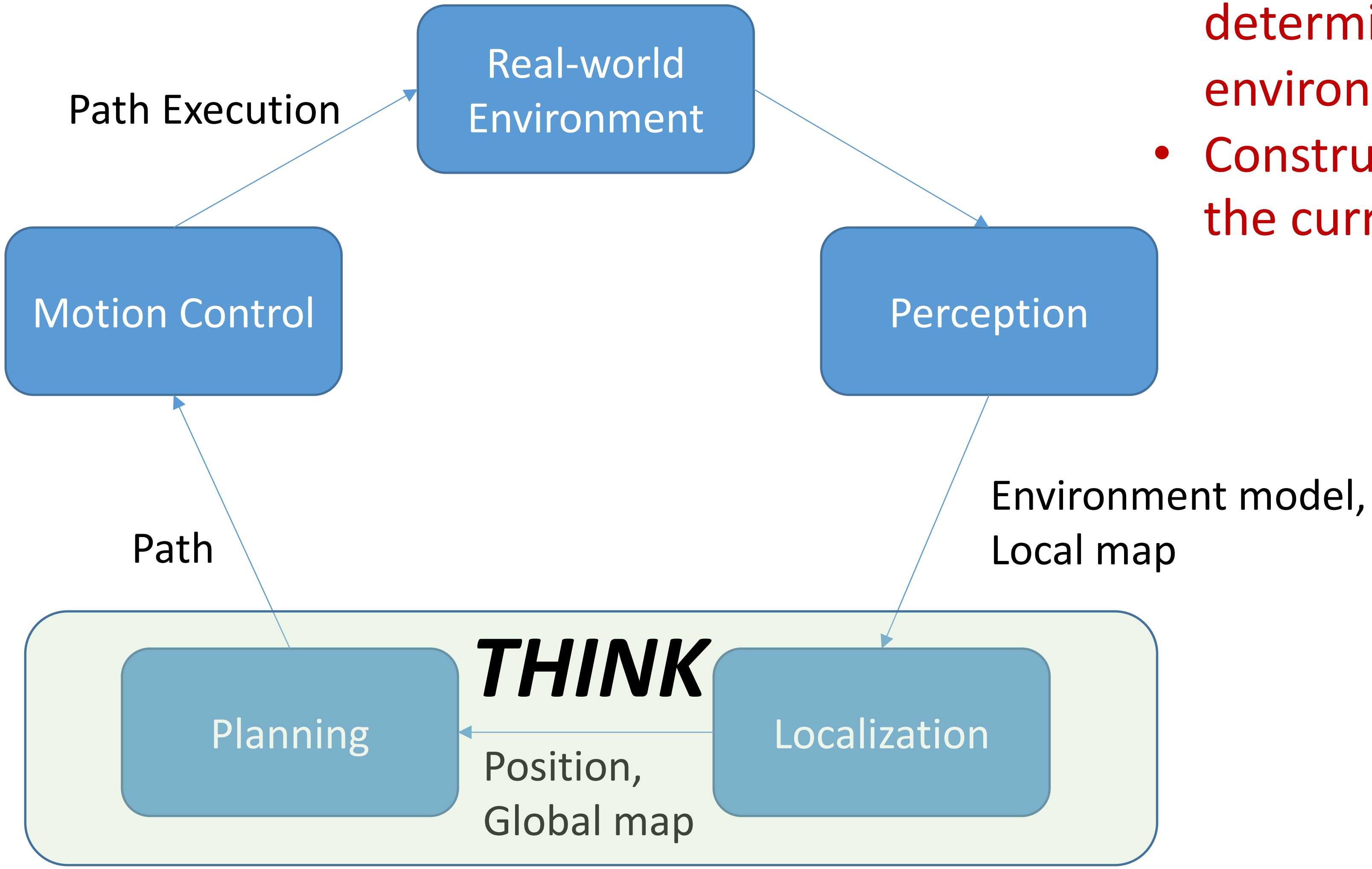
- We saw this diagram in the last lecture.
- Let’s look at it again, in the context of sense, think, act.

Example: Navigation in a Known Environment



Sensing provides a connection between the real world and the robot's internal representation of the world.

Example: Navigation in a Known Environment



Real-world Environment

Path Execution

Motion Control

Perception

Path

Environment model, Local map

THINK

Planning

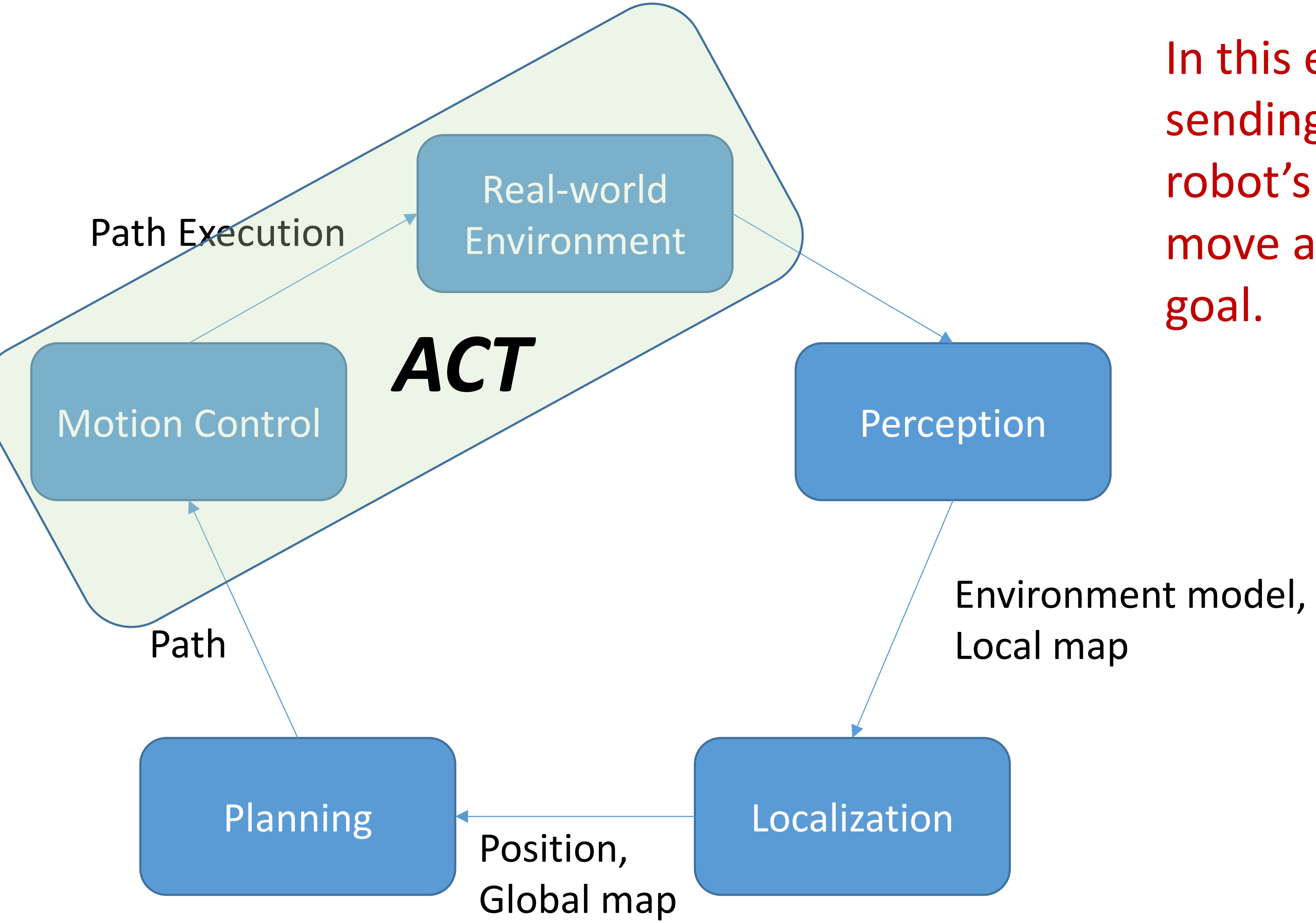
Localization

Position, Global map

In this example, *thinking* involves:

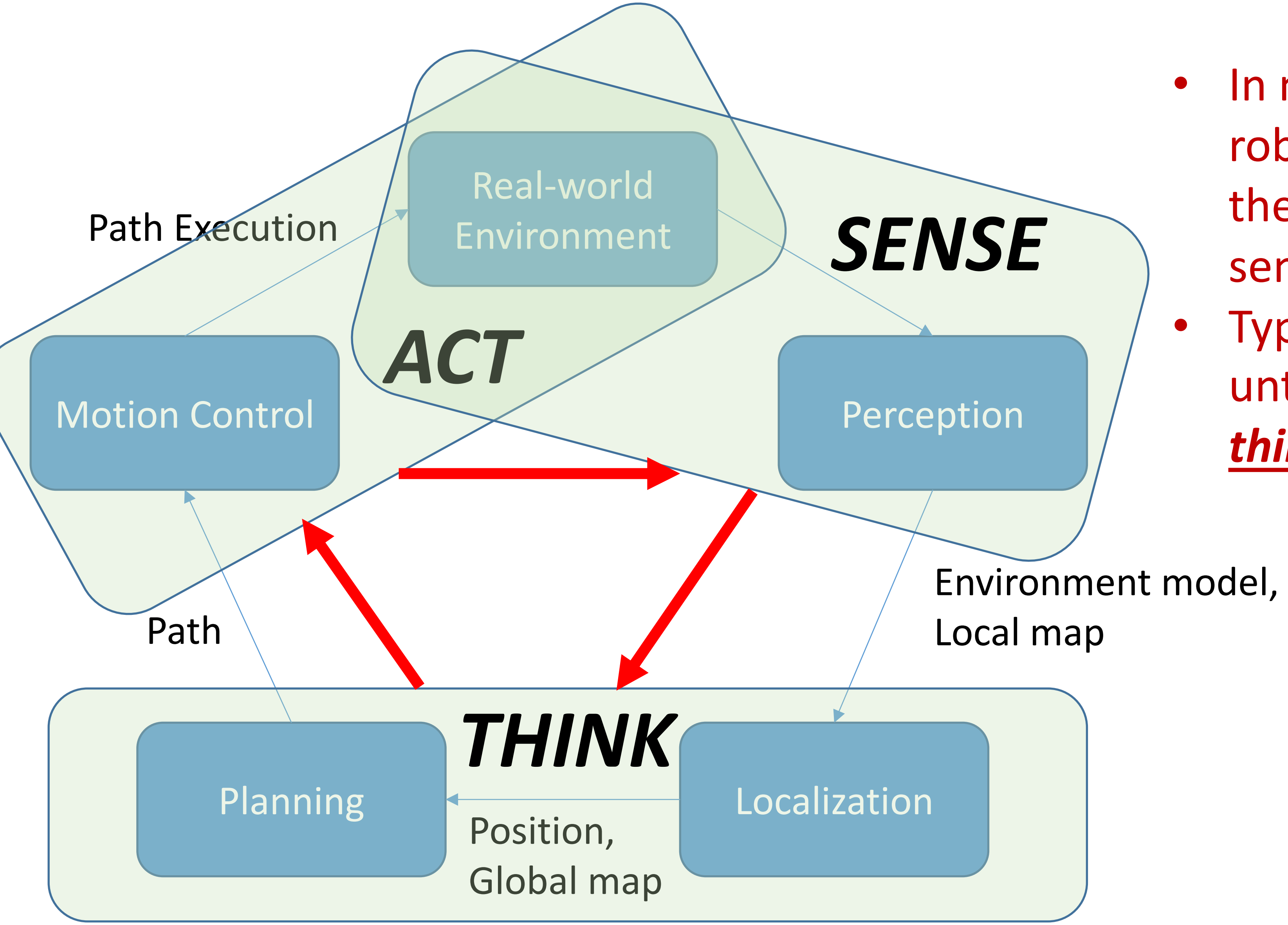
- Processing perceptual information to determine the position of the robot in its environment
- Constructing a motion plan to move from the current position to the goal position.

Example: Navigation in a Known Environment



In this example, *acting* involves sending motion commands to the robot’s motors, so that the robot will move along the desired path to its goal.

Example: Navigation in a Known Environment

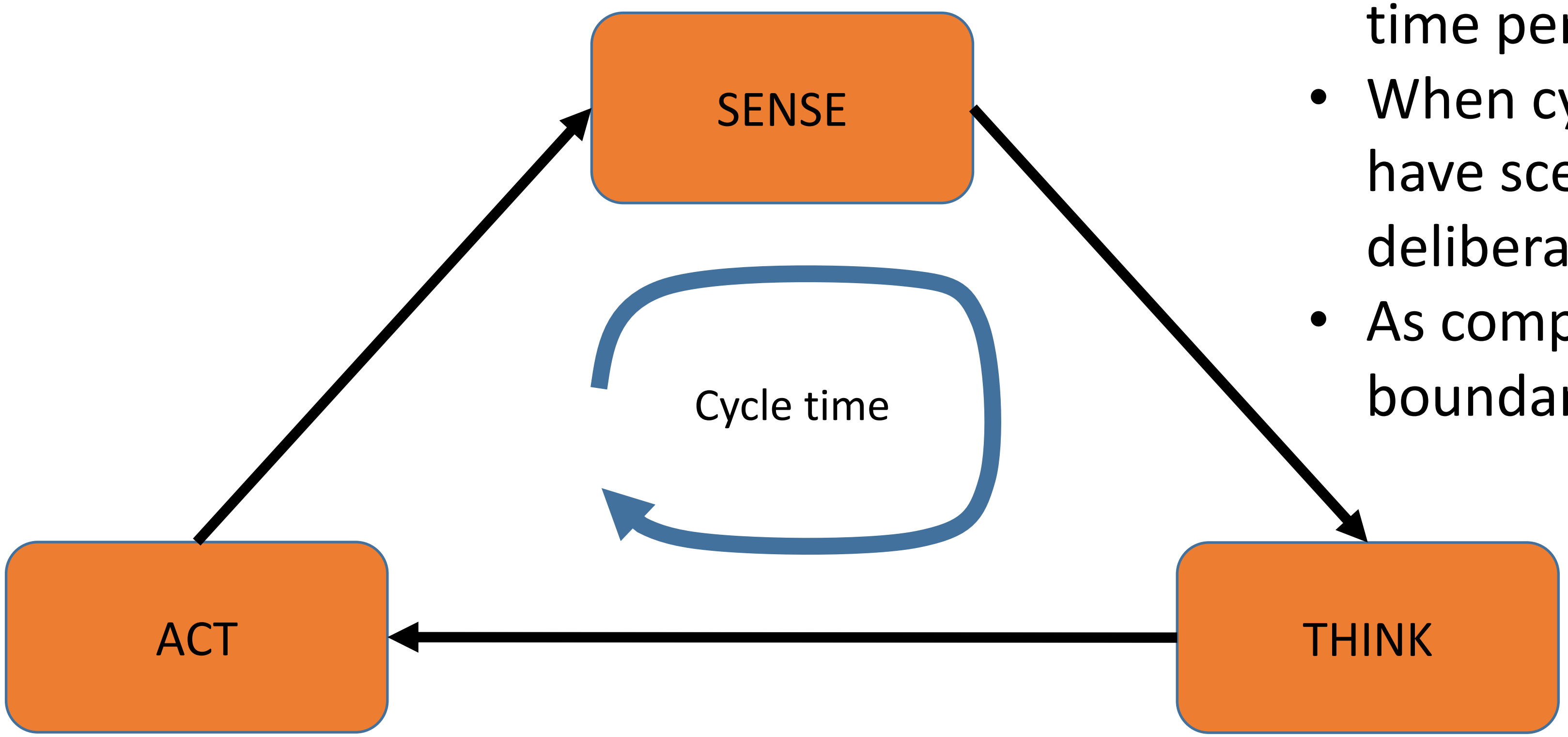


- In most robotics applications, the robot does not succeed to perform the task using a single episode of sense, think, act.
- Typically, these stages are repeated until the task is achieved: the ***sense, think, act loop.***

Sense, Think, Act at Different Time Scales

The time to complete one cycle of this loop depends on the task:

- Playing chess: minutes
- Hand-eye coordination: 30 Hz
- Force controlled robot: Order of KHz



- When cycle time is very fast, we use tools from control theory, and model systems using differential equations (continuous time performance).
- When cycle time is very slow, we might have scene understanding and deliberative planning.
- As computers become faster, the boundary between these begins to blur.

Representing the World

- Perception has the responsibility of converting sensor measurements into a representation of the world.
- Planning uses these representations to reason about the effects of actions in the world.

This raises the question:

What kind of representations should the robot use?

Symbolic Representations

For high-level task planning, it is often sufficient to represent the world using symbolic descriptions.

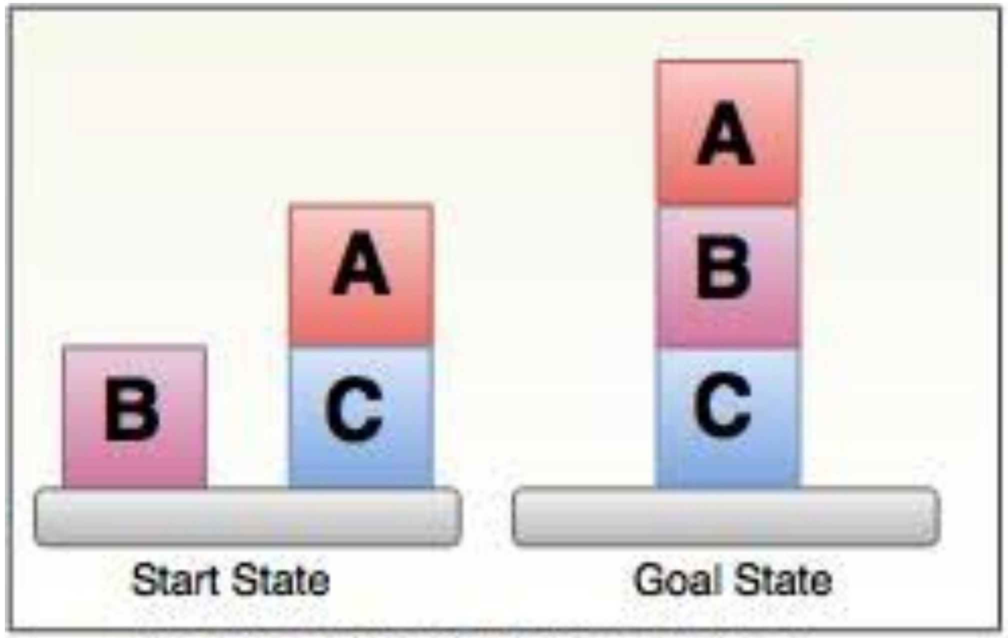


Fig: Blocks-World Planning Problem

Representation of Blocks World using simple predicates

Initial State:

- ON(table,B)
- On(table,C)
- On(A,C)
- Clear(B)
- Clear(A)

Goal State:

- ON(table,C)
- On(A,B)
- On(B,C)
- Clear(A)

High-Level Planning

A high-level planner uses a symbolic representation of actions:

- Preconditions: what must be true in the world before the action is applied?
- Effects: what changes occur in the world after the action occurs?

Pickup(?X):
  Preconditions: Gripper(empty)
  Effects: Gripper(full), Holding(?X)

If the goal is to be holding Block B, the planner can instantiate the variable ?X to B

Pickup(B):
  Preconditions: Gripper(empty)
  Effects: Gripper(full), Holding(B)

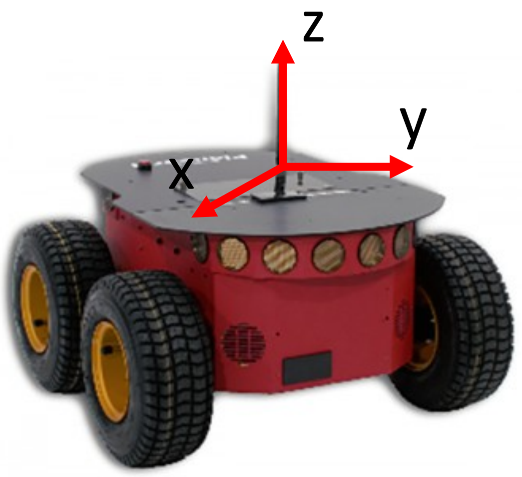
Geometric Representations

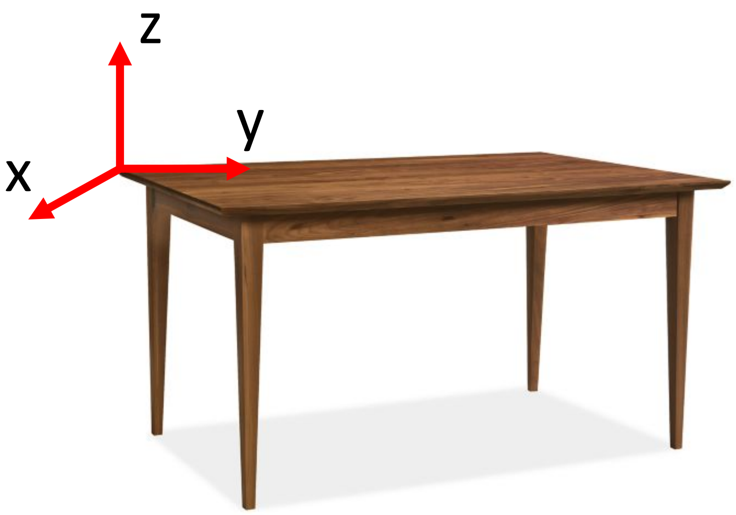
In robotics, we often require specific geometric information.

To describe an object's position:

- Attach a coordinate frame to the object (rigid attachment of frame to the object)
- Specify the position and orientation of the coordinate frame.

If we know this information, we know everything about the object's position!

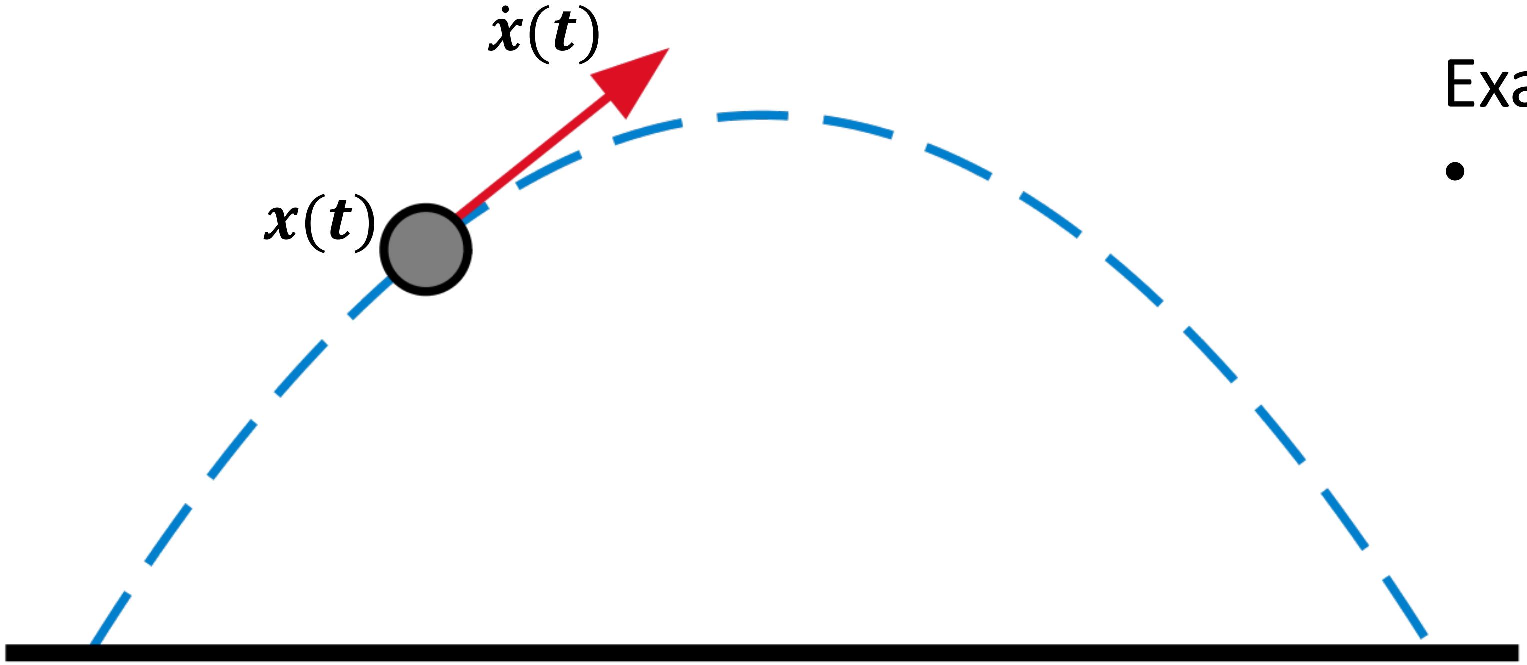




State

The term ***state*** is used in the study of dynamical systems to describe the relevant aspects of an objects motion.

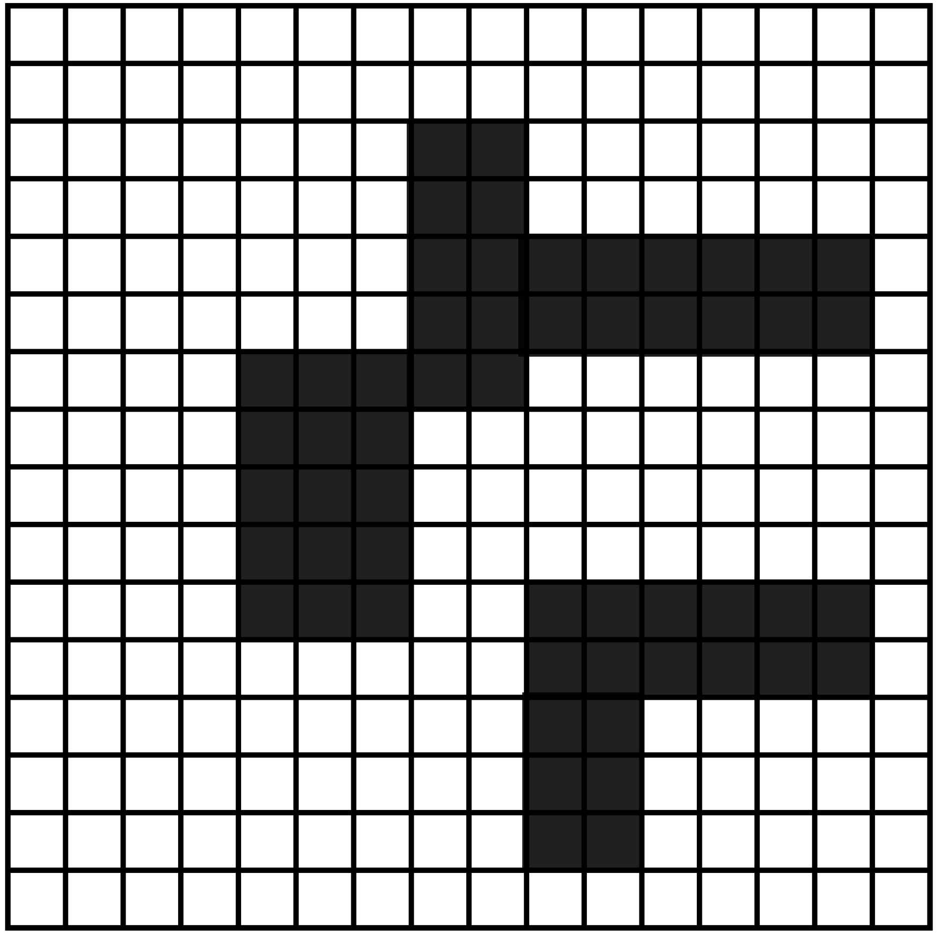
If we know the state x at time t_0 along with the system input for all $t \geq t_0$, then we can predict the state at all future times.



Example:

- If we know the position and velocity of a projectile at a given time, we can compute its entire trajectory.

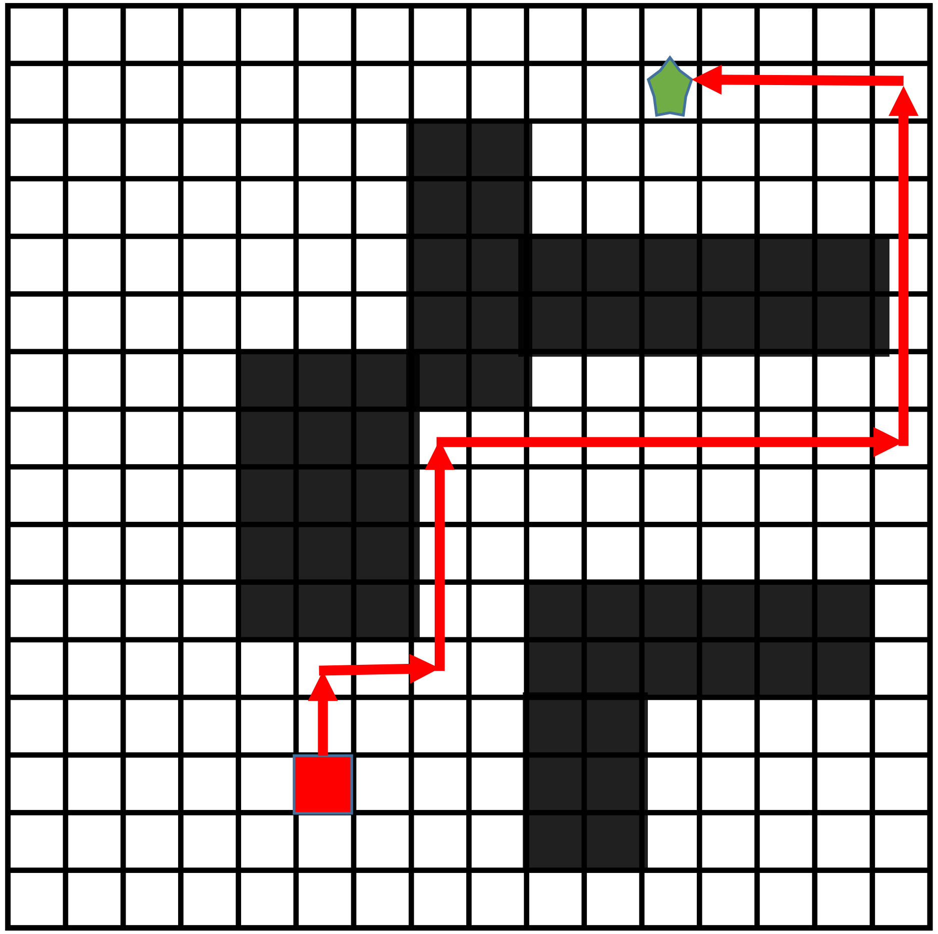
Grid World



- For many mobile robotics applications, one can represent the world as a grid.
- Each grid cell is either free or occupied by an obstacle.
- The path planning problem is to find a free path from start to goal.
- There are many variations, e.g., assign to each cell in the grid a *probability* that it is occupied by an obstacle (we’ll see this later).

Path Planning in a Grid World
The Simplest case of *Thinking*

Grid World: Path Planning



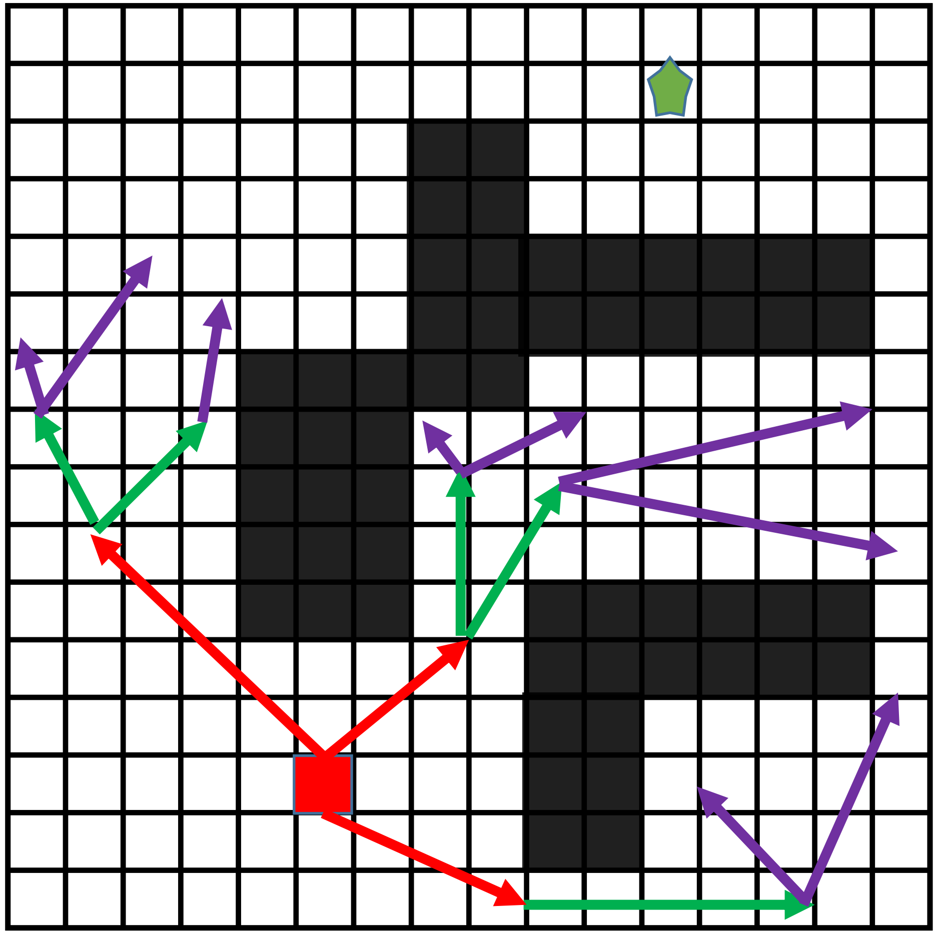
🟥 **Start position**

⬟ **Goal position**

One possible solution path.

- How can we effectively find any path from start to goal?
- How should we decide which path to take?

Grid World



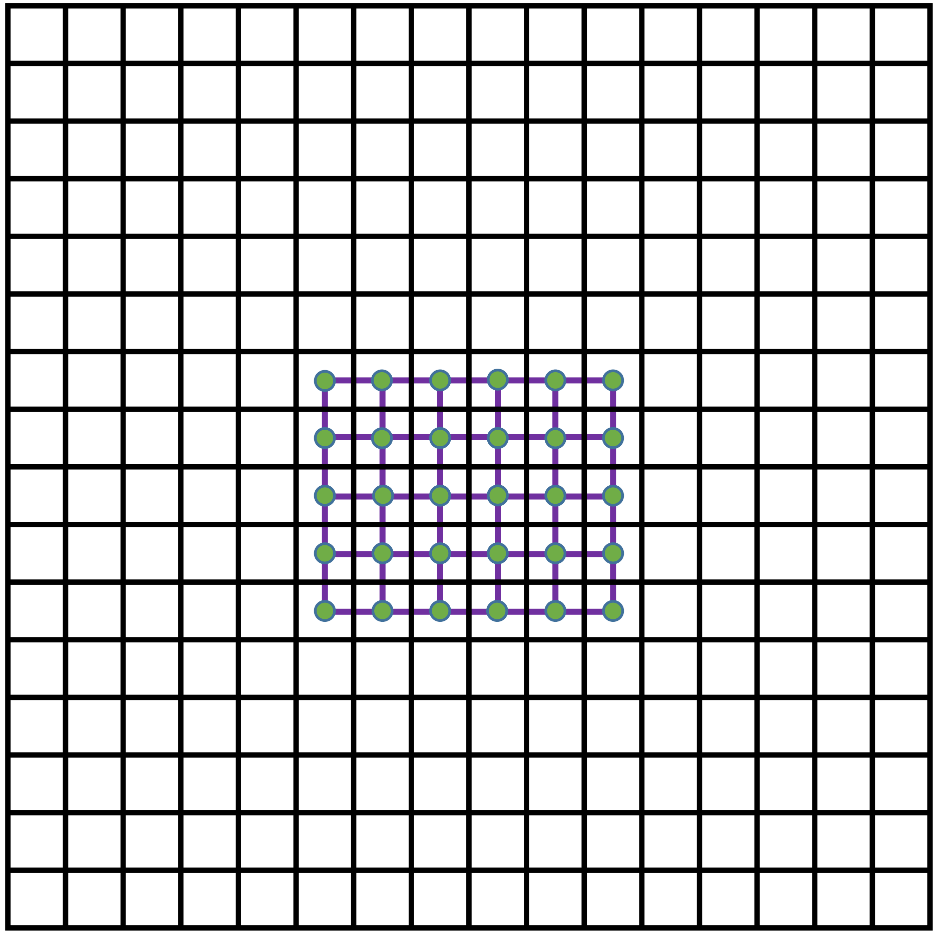
🟥 **Start position**

⬟ **Goal position**

One strategy is to systematically explore various possible solution paths.

This raises the question:
What strategies should we use to explore alternative paths?

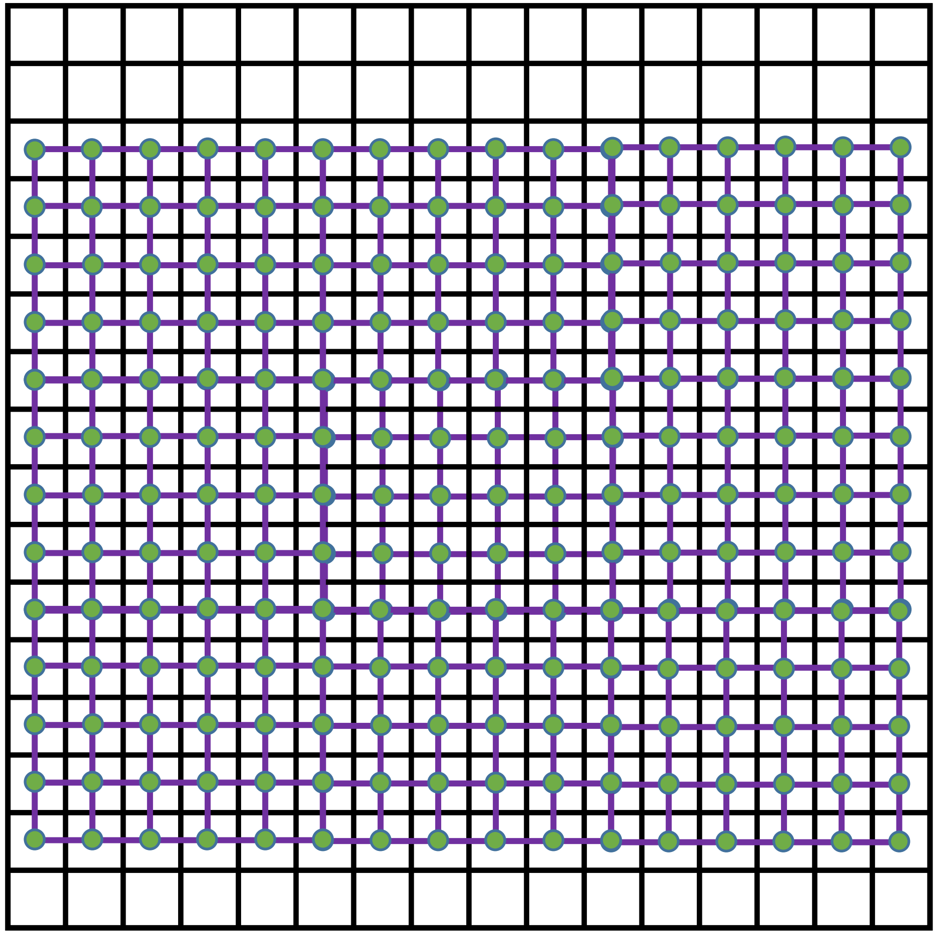
Grid World



A grid can be represented as a graph:

- Each cell in the grid corresponds to a vertex in the graph
- Vertices that correspond to adjacent grid cells are connected by an edge.

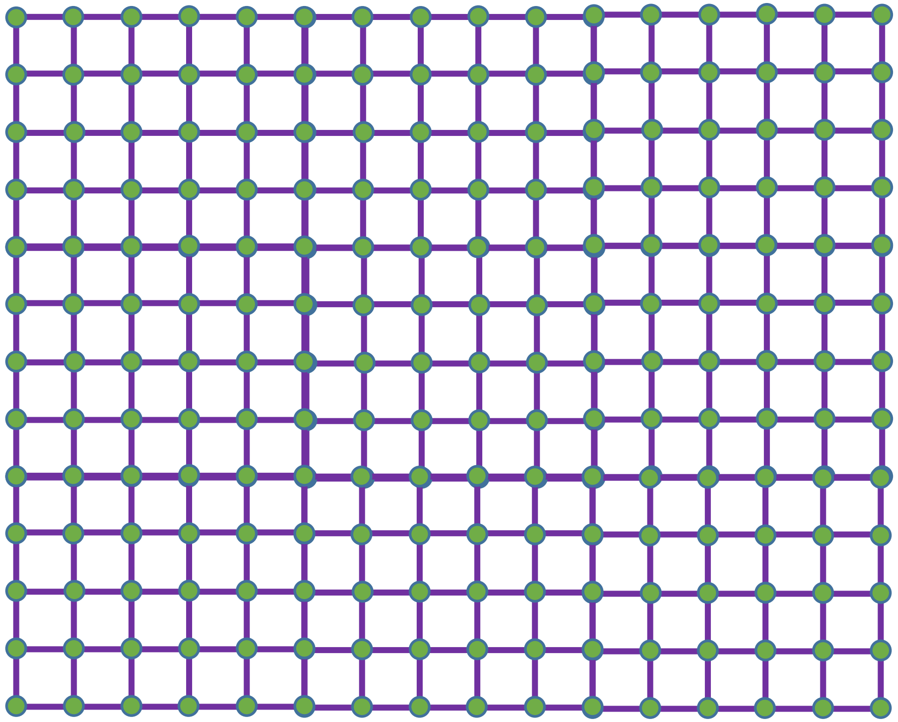
Grid World



A grid can be represented as a graph:

- Each cell in the grid corresponds to a vertex in the graph
- Vertices that correspond to adjacent grid cells are connected by an edge.

Grid World



A grid can be represented as a graph:

- Each cell in the grid corresponds to a vertex in the graph
- Vertices that correspond to adjacent grid cells are connected by an edge.

And now, we can use graph search algorithms to find a path!

Graph Traversal

- Problem: Find a path from a start vertex to a goal vertex
- Optional requirements:
    - Must traverse through certain nodes
    - Shortest path
    - Find one of multiple goals

- Solution: use search algorithms.

Tree Search

General Search Process

1. Check: did we run out of options? If so, planning failed.
2. Check: are we at the goal? If so, planning succeeded, return a path.

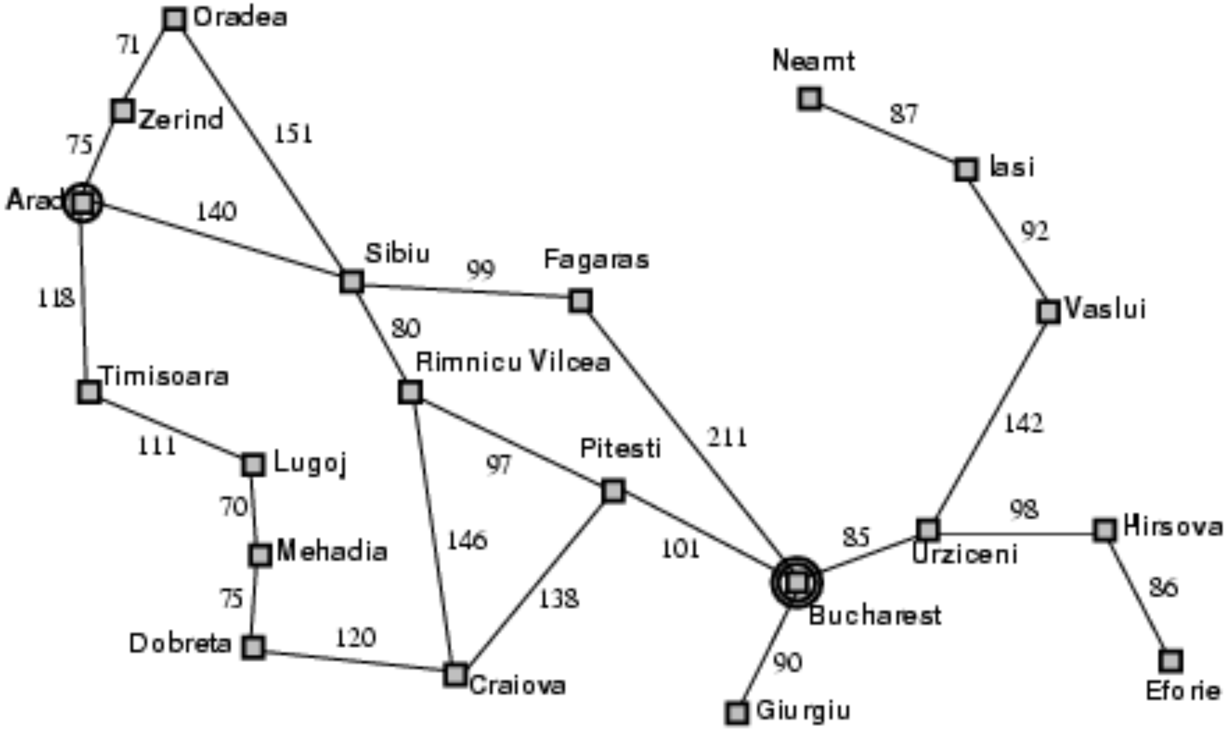
3. **Expand** the current state by considering each legal action (discovering the neighbors in the graph), thereby generating a new set of states. Keep these in a list (frontier)

   Note: all this planning happens in the robot’s “brain”, no actions are actually taken

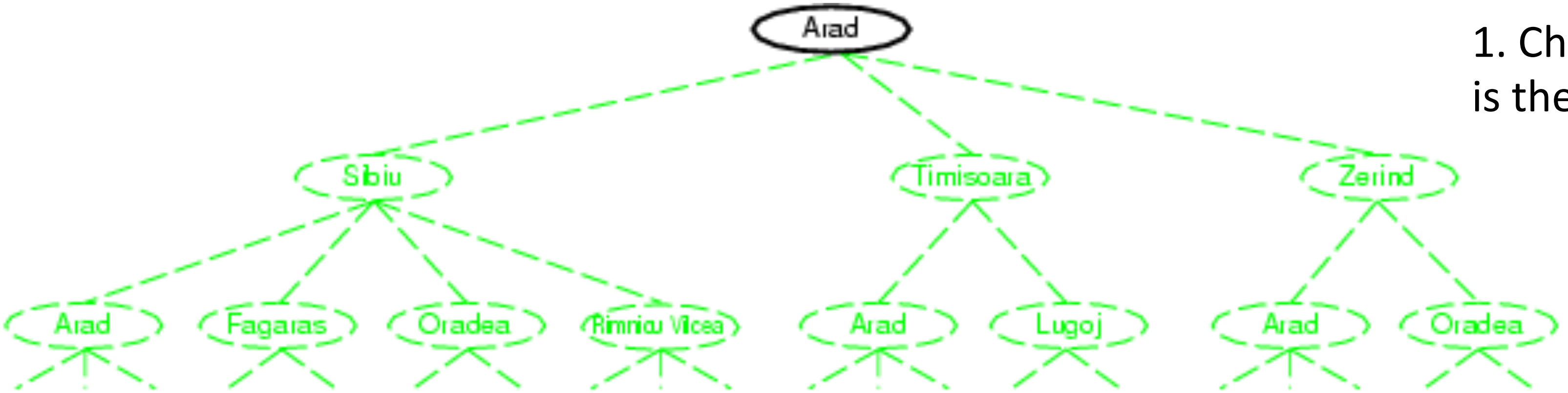
4. Simulate one of the possible actions from this list

5. Then go back to Step 1 and repeat.

Borrowing an example from AI: map of Romania

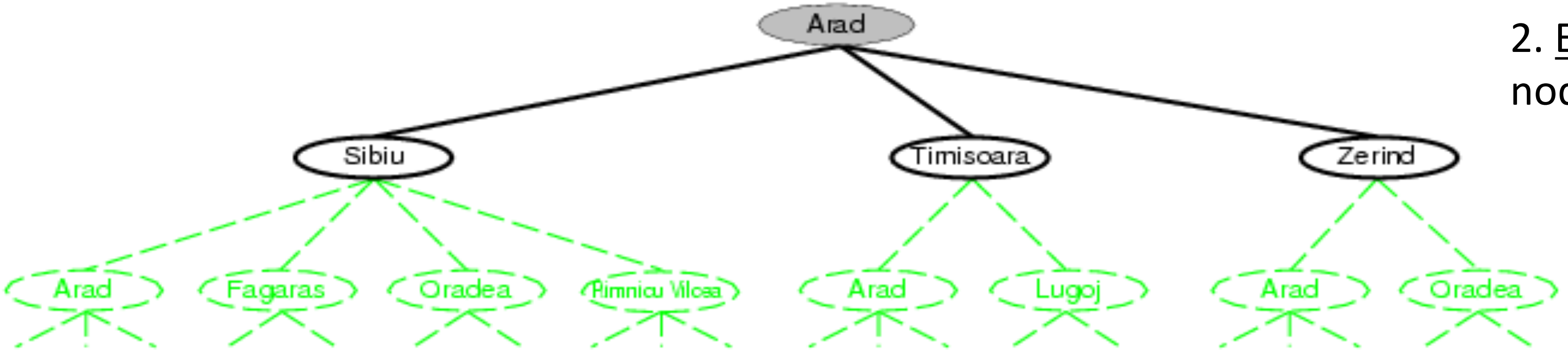


Tree search example



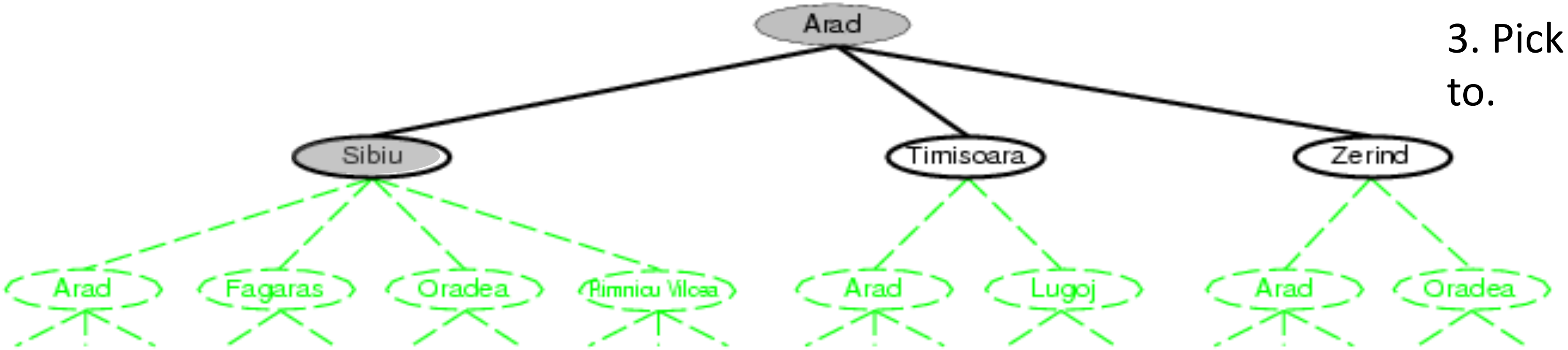
1. Check if current node is the goal

Tree search example



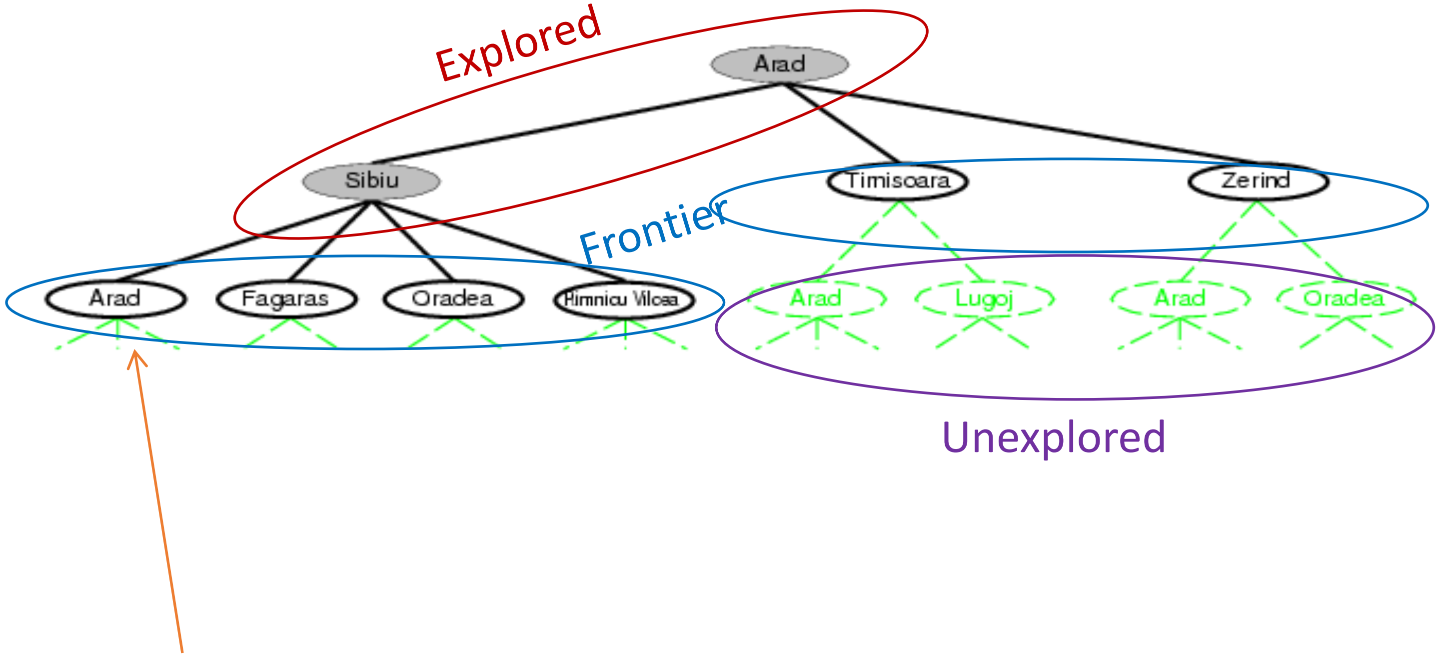
2. <u>Expand</u> neighboring nodes

Tree search example



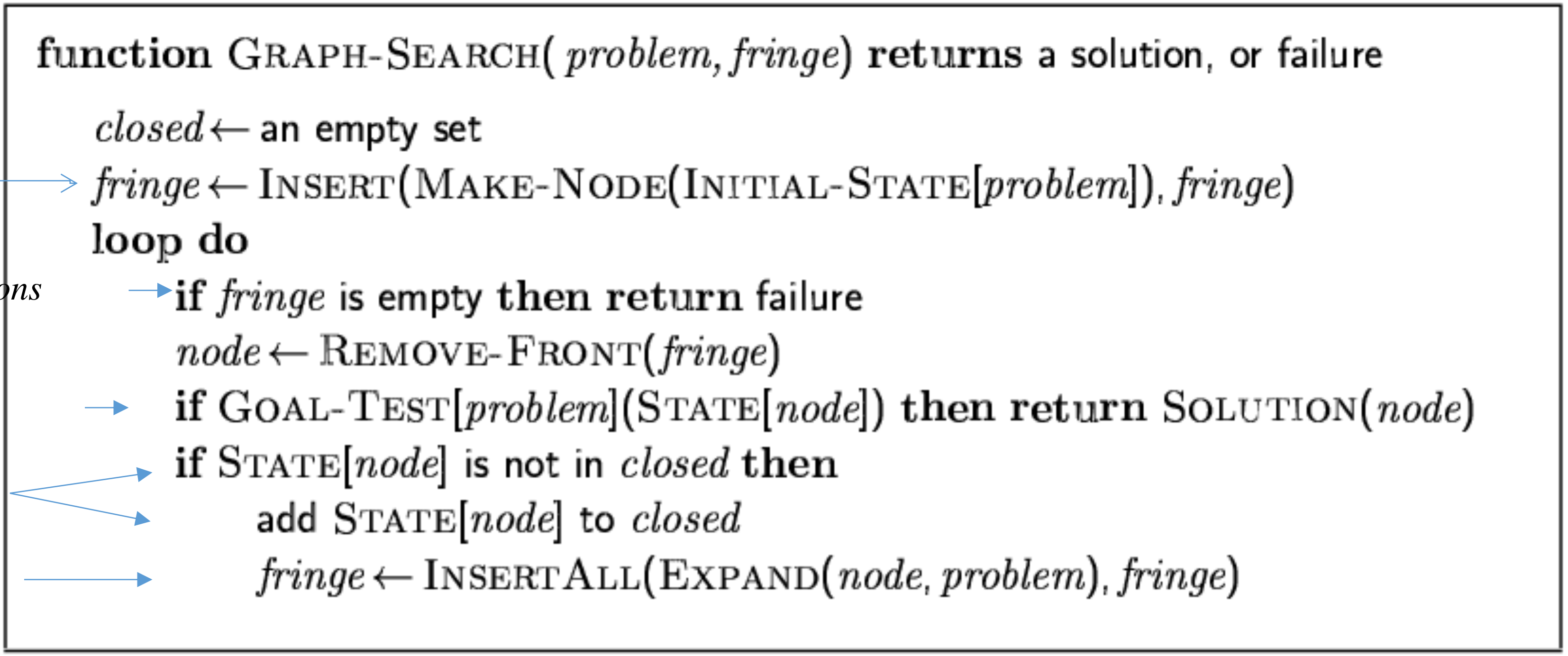
3. Pick a new node to go to.

Tree search example



Note that we could loop back to Arad. Have to make sure we don't go in circles forever!

Pseudocode



```
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 SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

a.k.a. frontier

Check if we ran out of options

Check if we're at the goal

(ensure we don't loop)

Expand node

Search strategies

- A search strategy is defined by picking the order of node expansion
    - Search algorithms differ mostly in the order in which they pick the nodes from the frontier

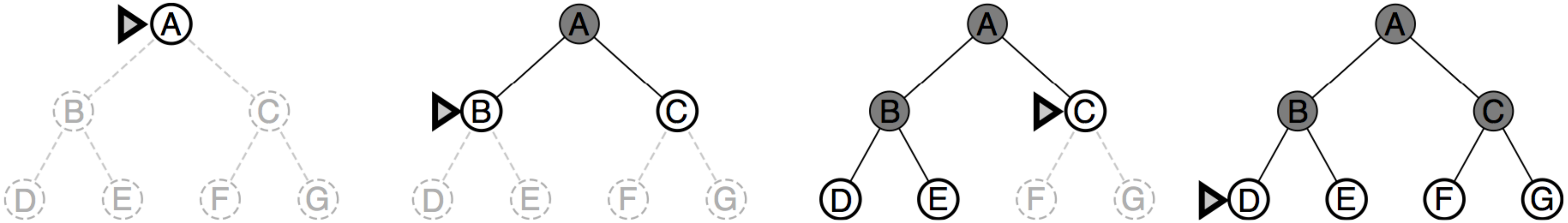
Uninformed search strategies

- Uninformed search strategies use only the topology of the graph: which states are connected by which actions. No additional information.

- Later we'll talk about informed search, in which you can estimate which actions are likely to be better than others.

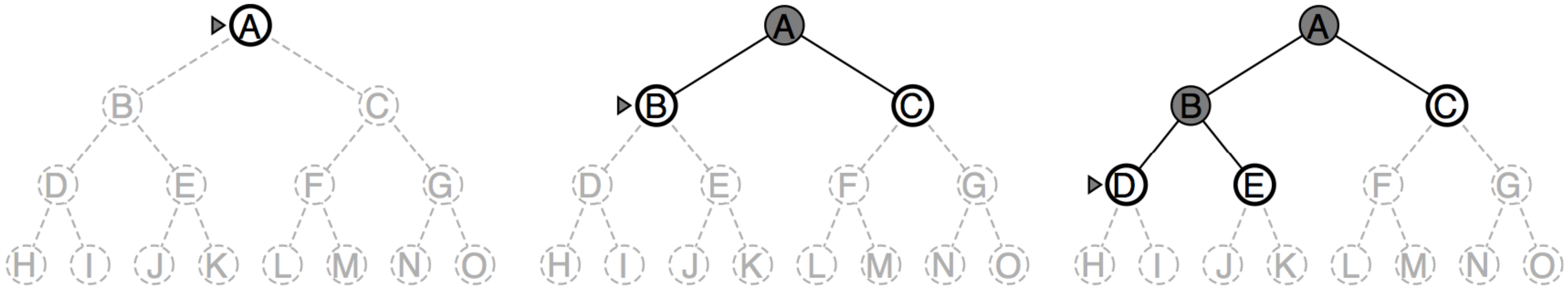
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *Frontier* is a FIFO queue, i.e., new successors go at end



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *Frontier* is a LIFO queue, i.e., put successors at front (i.e. a stack)



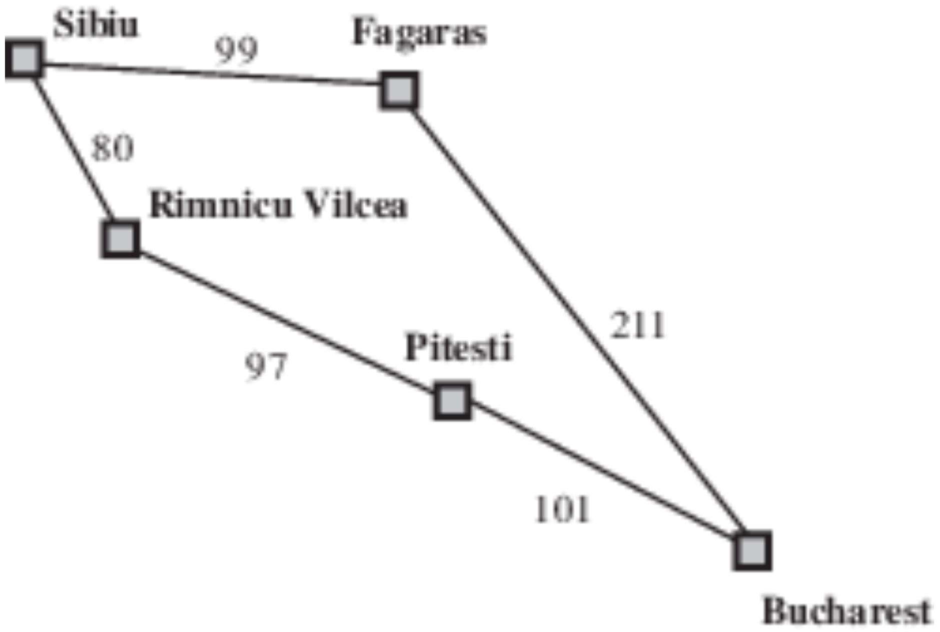
Comparison of BFS/DFS

- Breadth First Search and Depth First Search rely only on the structure of the graph
- BFS:
  - Guaranteed to find shortest path
  - Huge memory requirements
    - BFS b=10 to depth of 10
      - 3 hours (kind of bad)
      - 10 terabytes of memory (really bad)
- DFS
  - Efficient memory requirements
  - Does not guarantee to find shortest path
  - Might not terminate

Action Cost…

- BFS/DFS do not take into account the cost of actions

- Action cost, $g(n)$, is the total cost of moving from the start location to node n



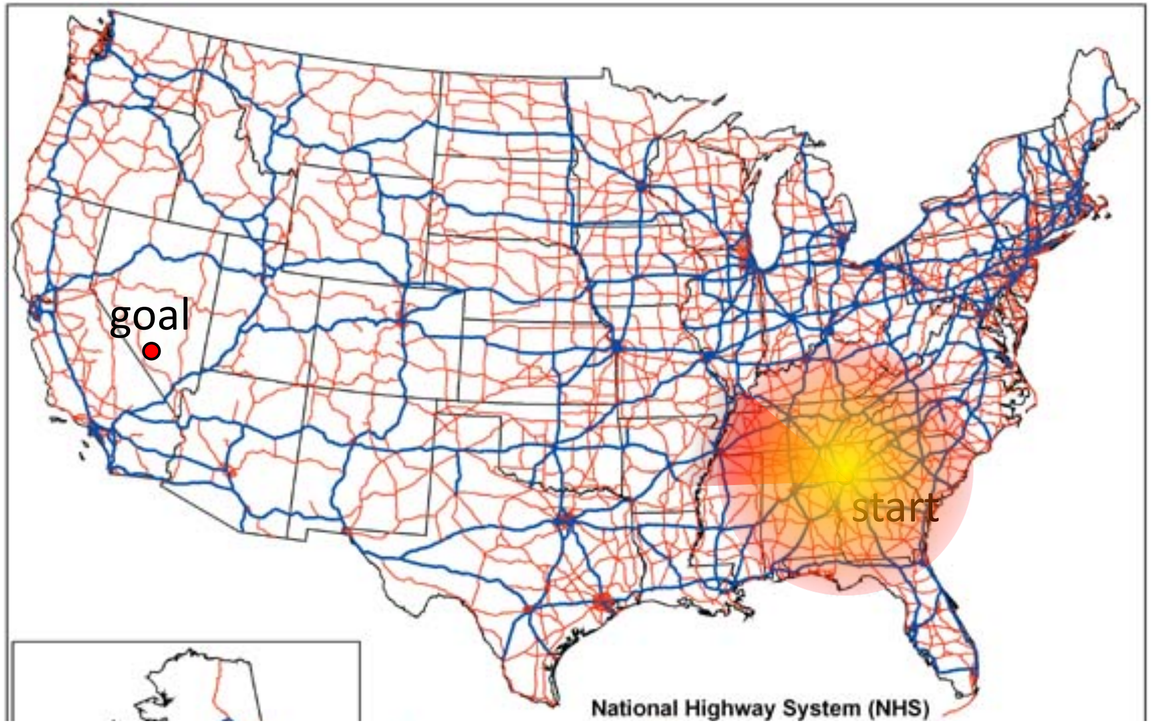
Uniform-cost search

- For graphs with actions of different cost
    - Equivalent to breadth-first if step costs all equal

- Expand least "total cost" unexpanded node
- Implementation:
    - *frontier*= queue sorted by path cost *g(n)*, from smallest to largest (i.e. a priority queue)

Note: Uniform Cost Search is same as Dijkstra's Algorithm, but focused on finding the shortest path to a single goal node rather than the shortest path to every node.

Informed Search



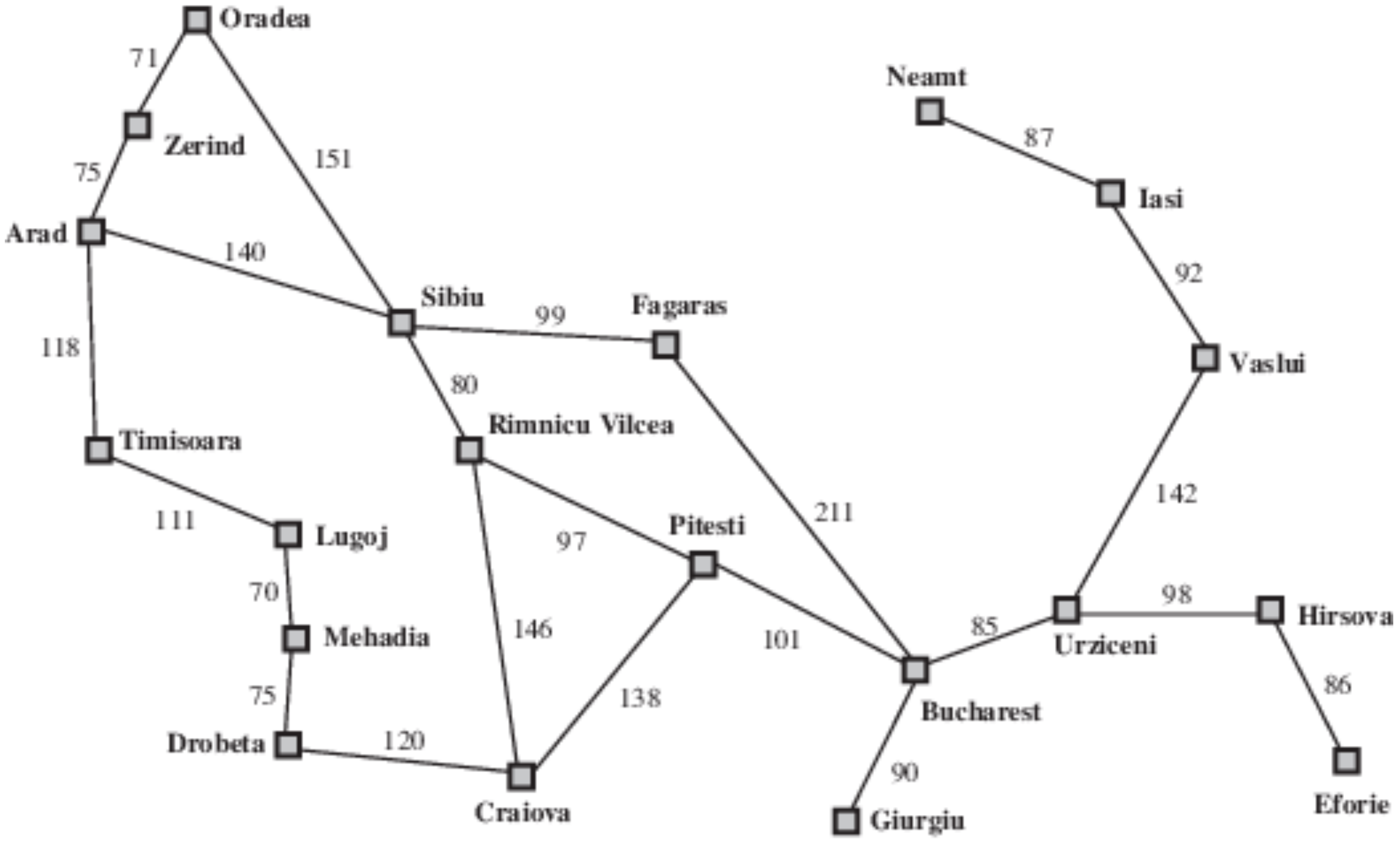
Uninformed search

Informed search

Informed Search

- What if we had an evaluation function $h(n)$ that gave us an estimate of the cost of how far n is from the goal
    - *h(n)* is called a *heuristic*

Romania with step costs in km



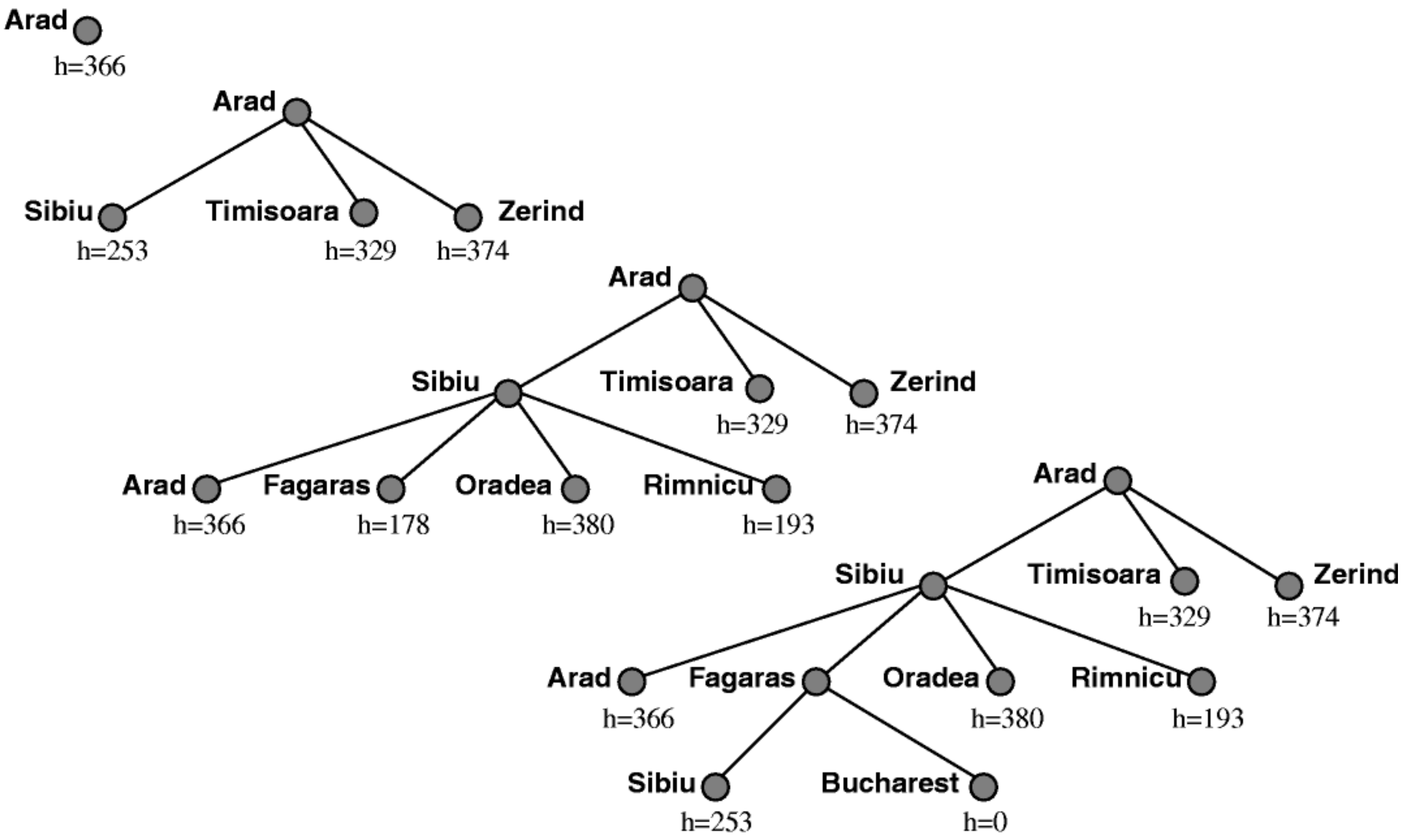
	h(n)
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic)
  - e.g., $f(n) = h_{SLD}(n)$ = straight-line distance from n to Bucharest

- Greedy best-first search expands the node that is estimated to be closest to goal

Best-First Algorithm



Performance of greedy best-first search

- Not guaranteed to find shortest path

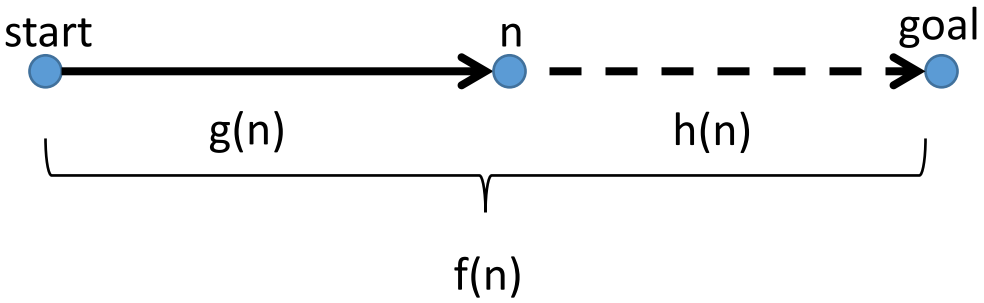
- With a good heuristic, it can be very efficient.

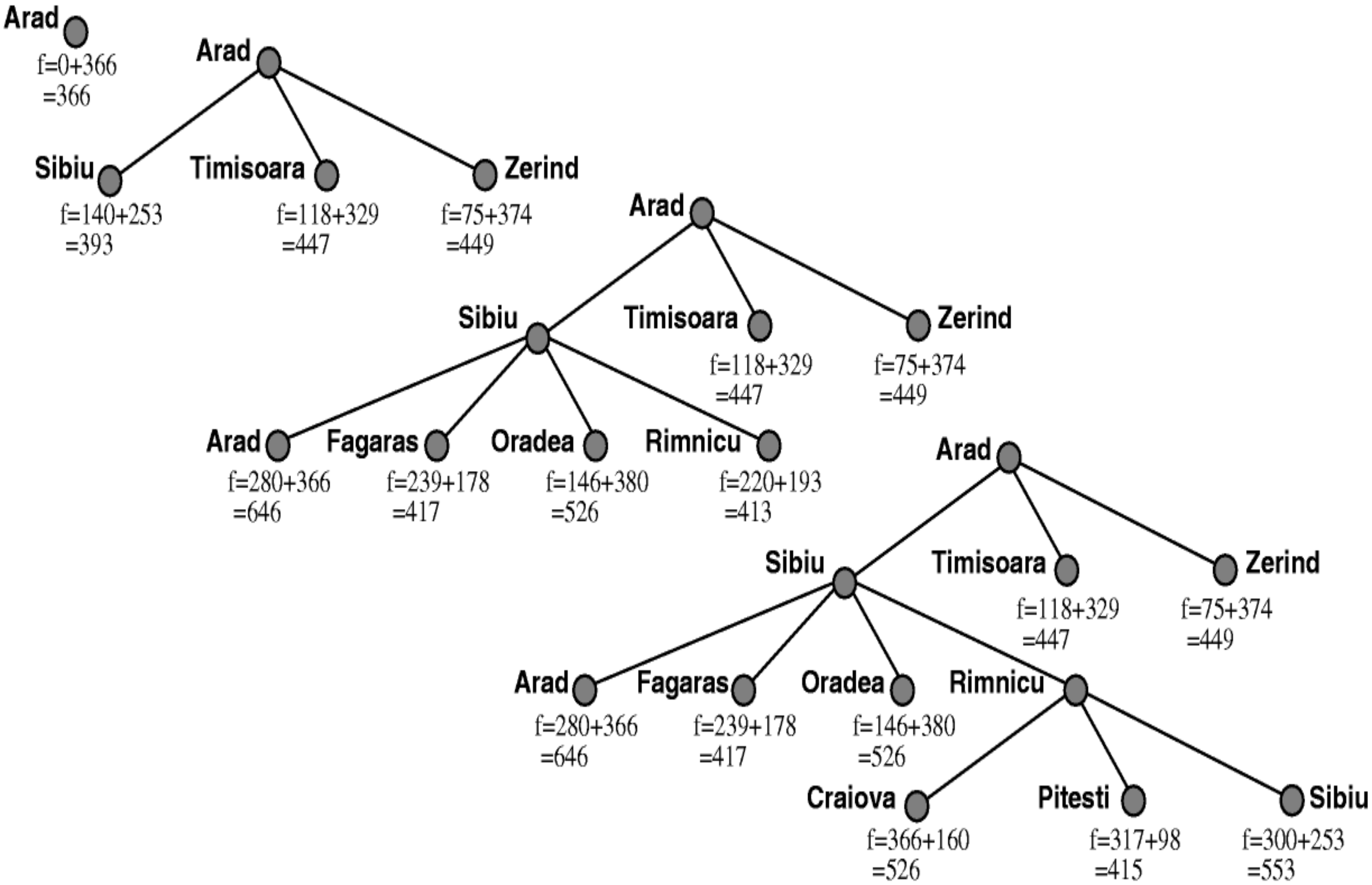
What can we do better?

A^* search

- Avoid expanding paths that are already expensive
- Consider
    - Cost to get here (known) – $g(n)$
    - Cost to get to goal (estimate from the heuristic) – $h(n)$

- Evaluation function *f(n) = g(n) + h(n)*
    - *g(n)* = cost so far to reach *n*
    - *h(n)* = estimated cost from *n* to goal
    - *f(n)* = **estimated total cost** of path through *n* to goal




Arad
f=0+366
=366
Arad
Sibiu
f=140+253
=393
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
Sibiu
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
f=280+366
=646
Fagaras
f=239+178
=417
Oradea
f=146+380
=526
Rimnicu
f=220+193
=413
Arad
Sibiu
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
f=280+366
=646
Fagaras
f=239+178
=417
Oradea
f=146+380
=526
Rimnicu
Craiova
f=366+160
=526
Pitesti
f=317+98
=415
Sibiu
f=300+253
=553

A* Heuristics

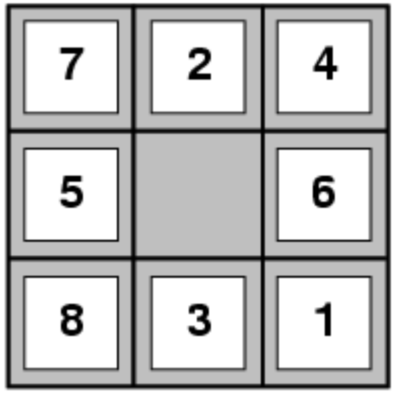
- A heuristic $h(n)$ is admissible if for every node n,

  $h(n) \leq h^{*}(n)$, where $h^{*}(n)$ is equal the true cost, $g^{*}(n)$, of reaching the goal state from n.

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
  - Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

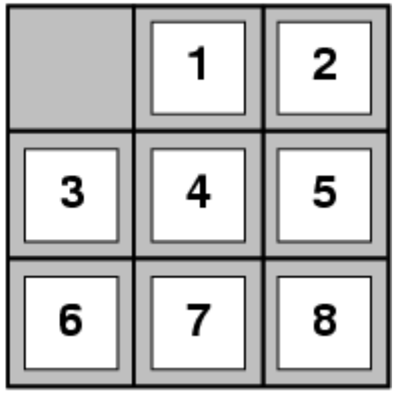
Admissible heuristics

E.g., for the 8-puzzle:



7	2	4
5		6
8	3	1

Start State



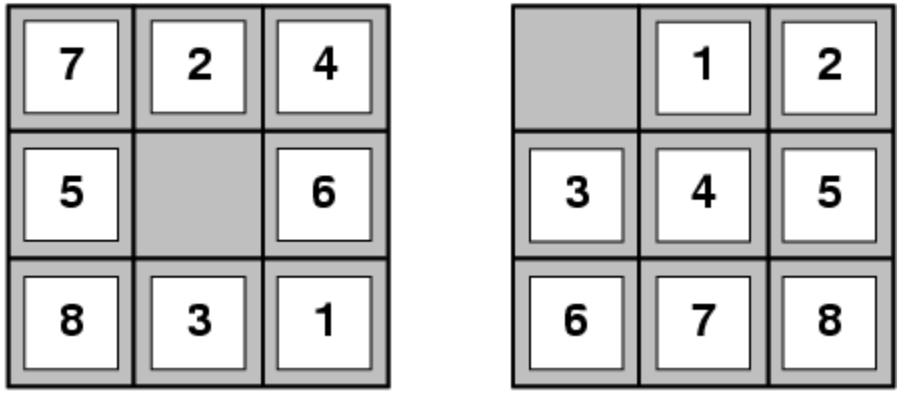
	1	2
3	4	5
6	7	8

Goal State

Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (i.e., number of squares from desired location of each tile)



7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

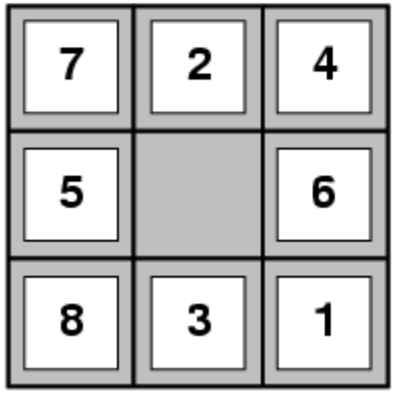
Goal State

- $h_1(S)$ = ?
- $h_2(S)$ = ?

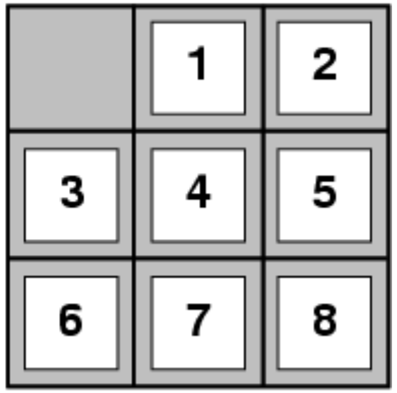
Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (i.e., number of squares from desired location of each tile)



Start State



Goal State

- $h_1(S)$ = ? 9
- $h_2(S)$ = ? 3+1+2+2+2+3+3+2 = 18

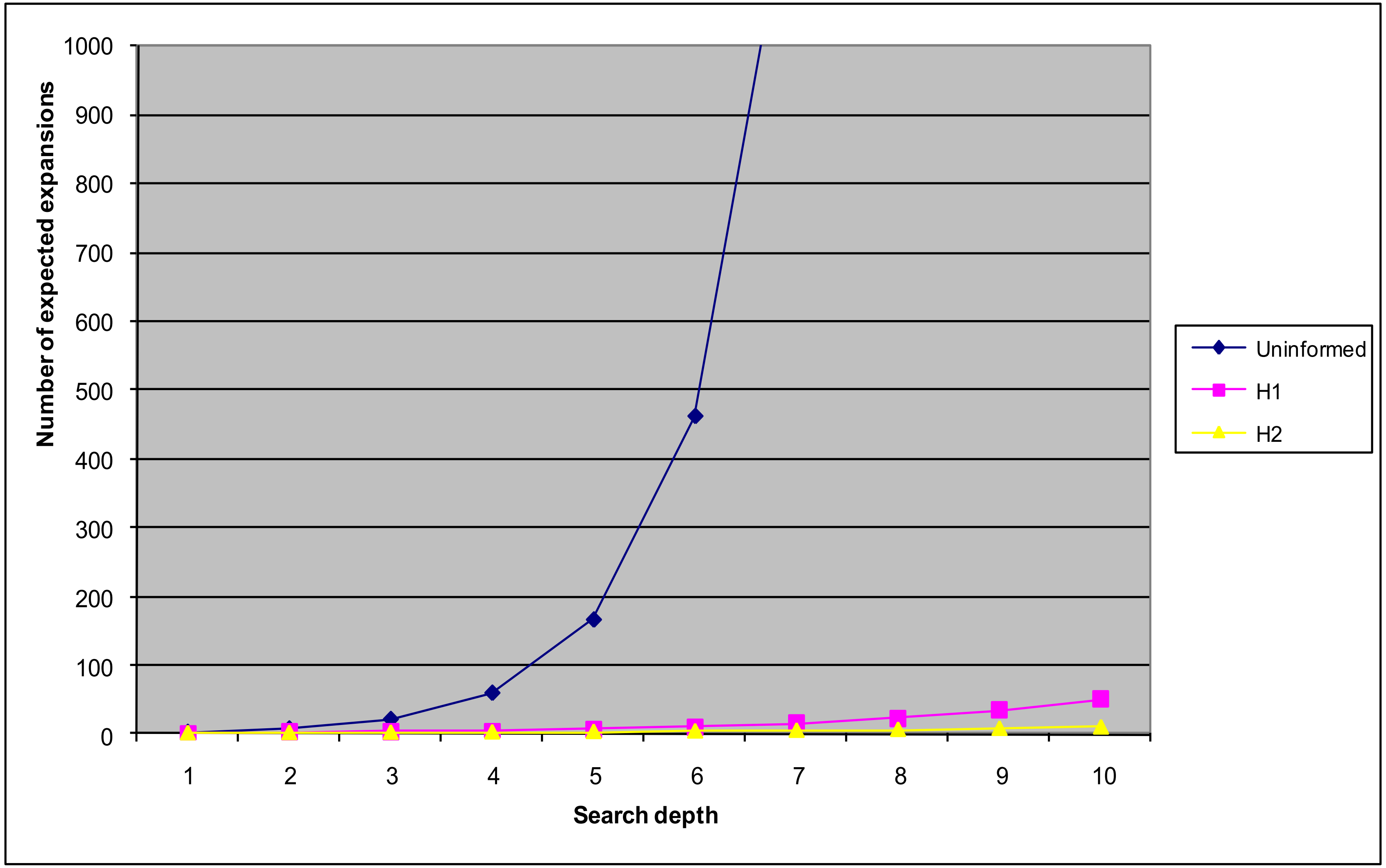
Which is better?

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
  - then h_2 dominates h_1
  - → h_2 is better for search

- What does better mean?
  - Finds the solution faster, expands fewer nodes

Visually



Number of expected expansions
1000
900
800
700
600
500
400
300
200
100
0
1
2
3
4
5
6
7
8
9
10
Search depth
Uninformed
H1
H2

What happens if heuristic is not admissible?

- Will still find a solution, but possibly not the optimal solution

The heuristic h(x) guides the performance of A*

- Let d(x) be the actual distance between S and G
  - h(x) = 0 :
    - A* is equivalent to Uniform-Cost Search
  - h(x) <= d (x) :
    - guarantee to compute the shortest path; the lower the value h(x), the more node A* expands
  - h(x) = d (x) :
    - follow the best path; never expand anything else; difficult to compute h(x) in this way!
  - h(x) > d(x) :
    - not guarantee to compute a best path; but very fast
  - h(x) >> g(x) :
    - h(n) dominates -> A* becomes the best first search

A* in Robotics

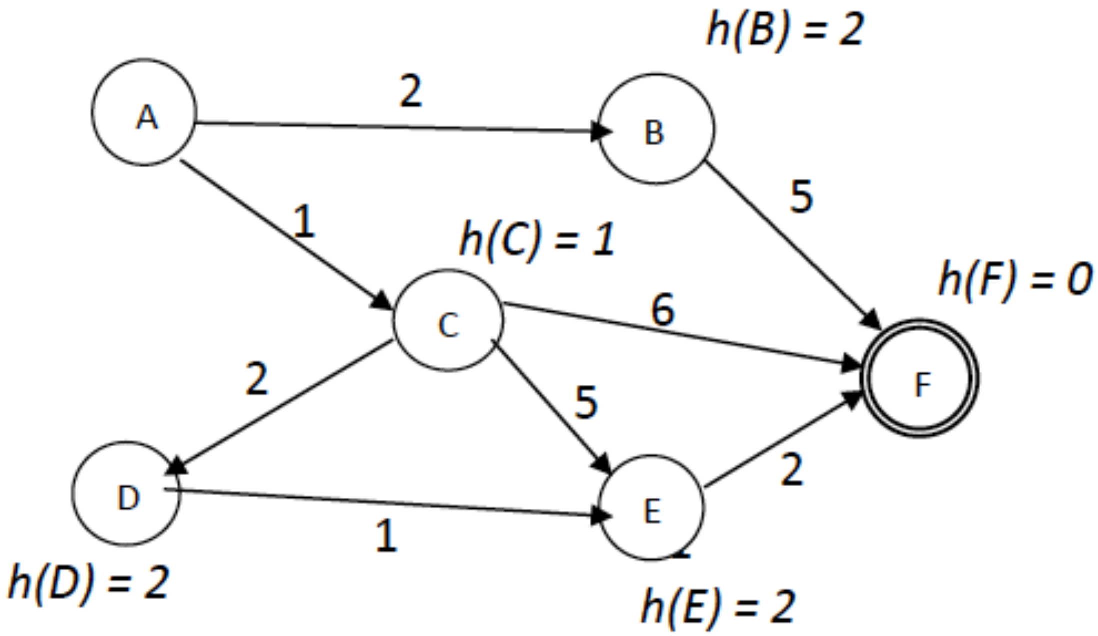
- One of the most frequently used algorithms for path planning, manipulation, and obstacle avoidance due to its efficiency.
- Primarily used in 2D environments.

Search Algorithm Summary

- Uninformed (topology only):
  - Breadth First Search *(does not consider path cost)*
  - Depth First Search *(does not consider path cost)*
  - Uniform Cost *(considers path cost g(n))*
- Informed:
  - Greedy Best-First Search *(heuristic h(n) only)*
  - A* Search *(h(n) + g(n))*

- *Any of these algorithms can be used to find a solution to the graphs below*

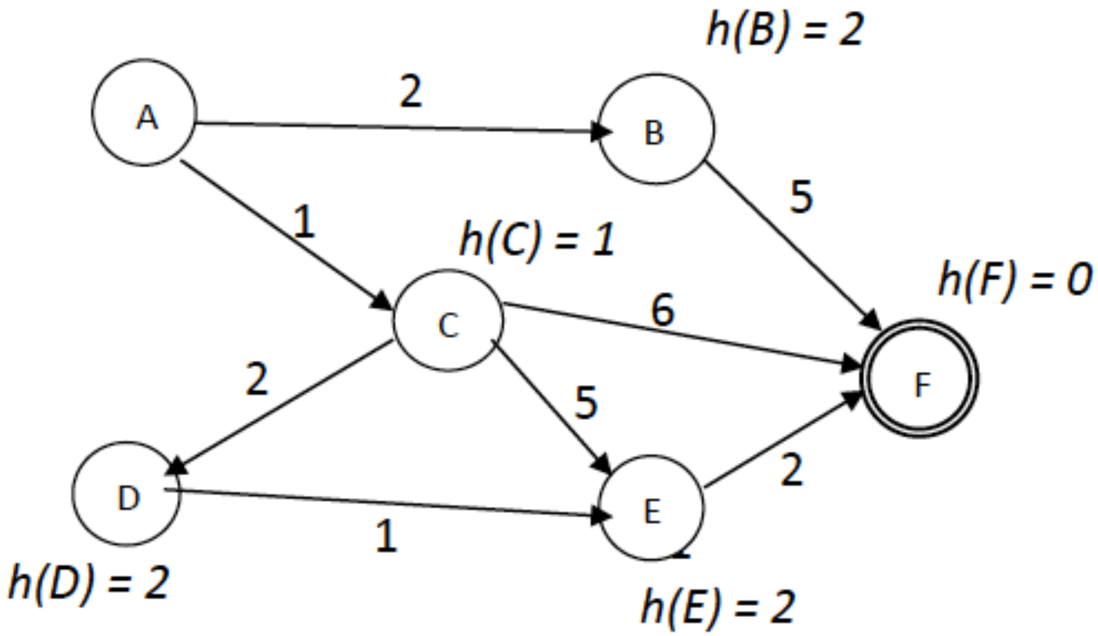
Practice A*



What is the order in which nodes are expanded if start is A and goal is F?

What is the final path from A to F?

Practice A*



What is the order in which nodes are expanded if start is A and goal is F?

ACBDEF

What is the final path from A to F?

ACDEF