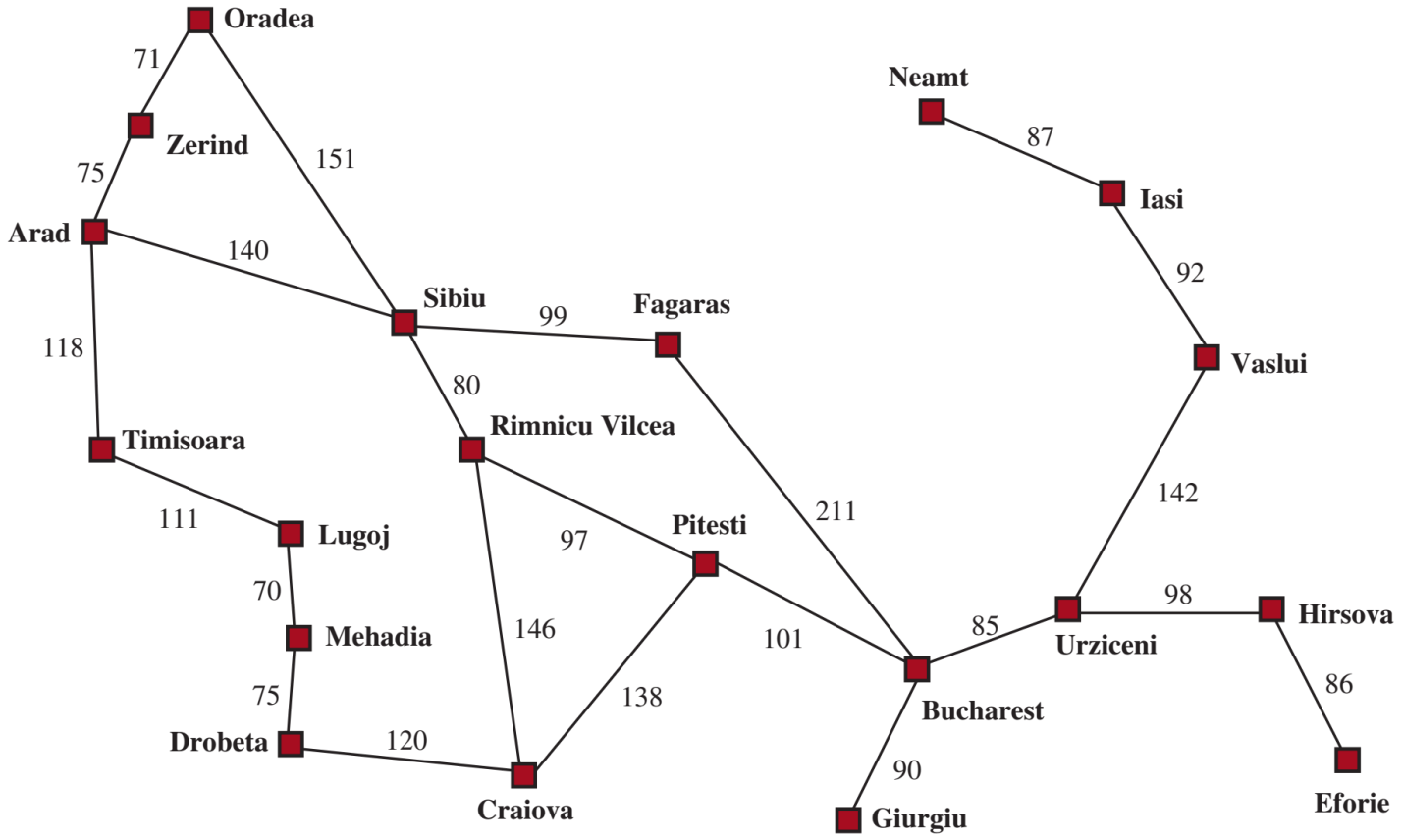
Lecture 4
Informed Search
Search in Complex Environments

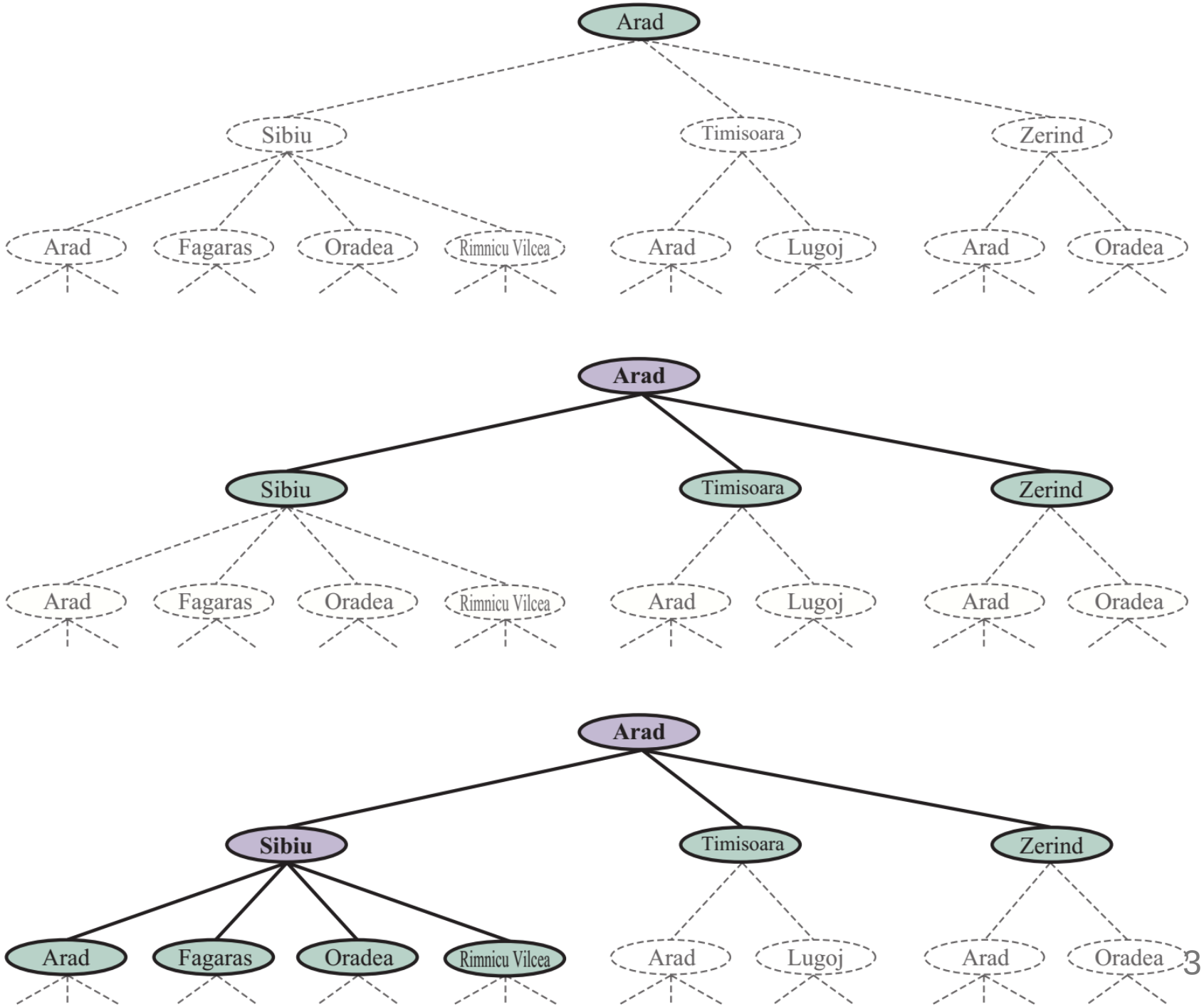
Gleb Sizov
Norwegian University of Science and Technology

Search problem

1. State-space
2. Initial state
3. Goal states
4. Actions
5. Transition model
6. Action cost function



Search trees



Uninformed vs informed search

Uninformed search - no clue about how close a state is to the goal(s).

Informed search - have some estimate of how close a state is to the goal(s).

S1

7	5	2
	4	3
8	1	6

S2

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7	8	

Goal state

Informed search - A*

Best-first search with $f(n) = g(n) + h(n)$

- $f(n)$ - estimated cost of the cheapest path through n to goal
- $g(n)$ - cost so far to reach node n
- $h(n)$ - estimated cost of the cheapest path from n to goal

Properties of A*

Complete:

Yes, for positive action costs, state space has a solution or is finite.

Cost optimal: Yes, when

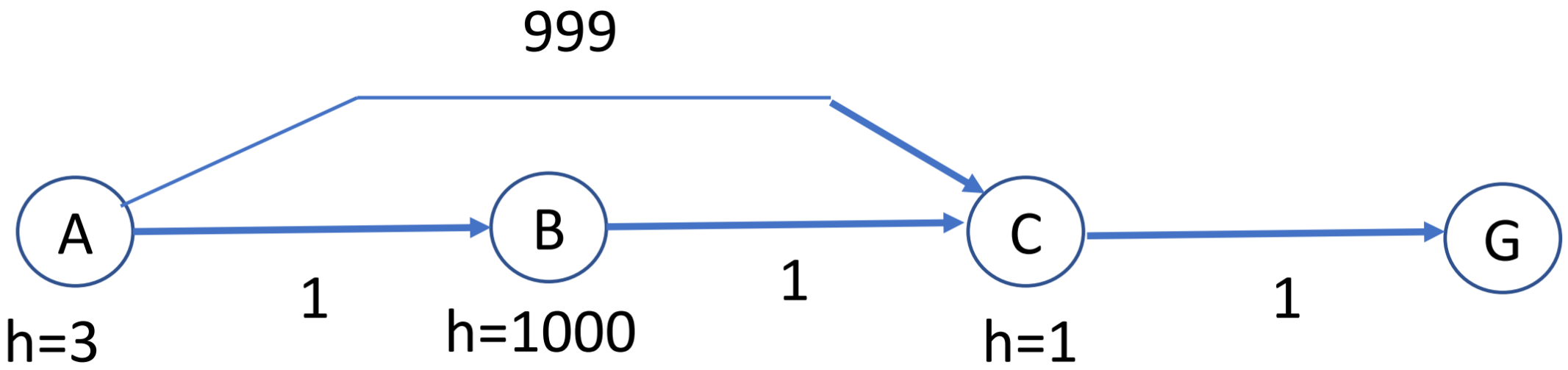
- Finite branching factor
- Costs are positive
- For tree search, h is admissible and non-negative
- Gor graph search, h is consistent and non-negative

Time and space: $O(b^{*d})$, for good heuristic $b^* < b$

Admissibility of h

Aadmissibility - optimistic, never overestimates the cost to reach a goal.

$h(n) < h^*(n)$



Admissible?

Optimal?

Consistency of h

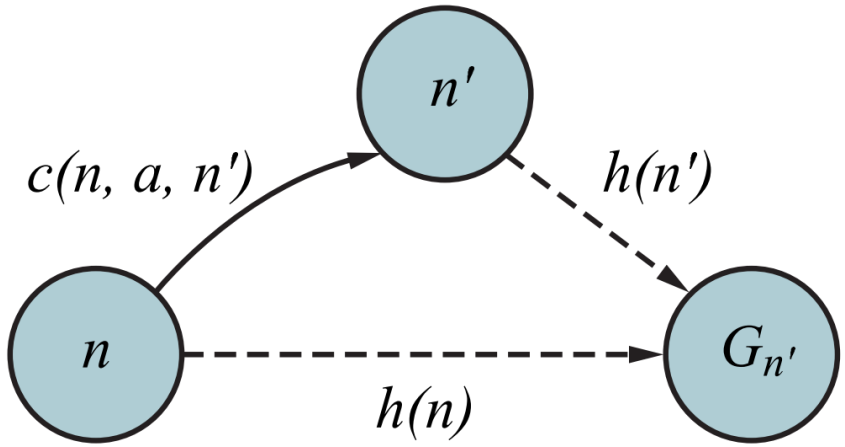
Consistency - triangle inequality

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

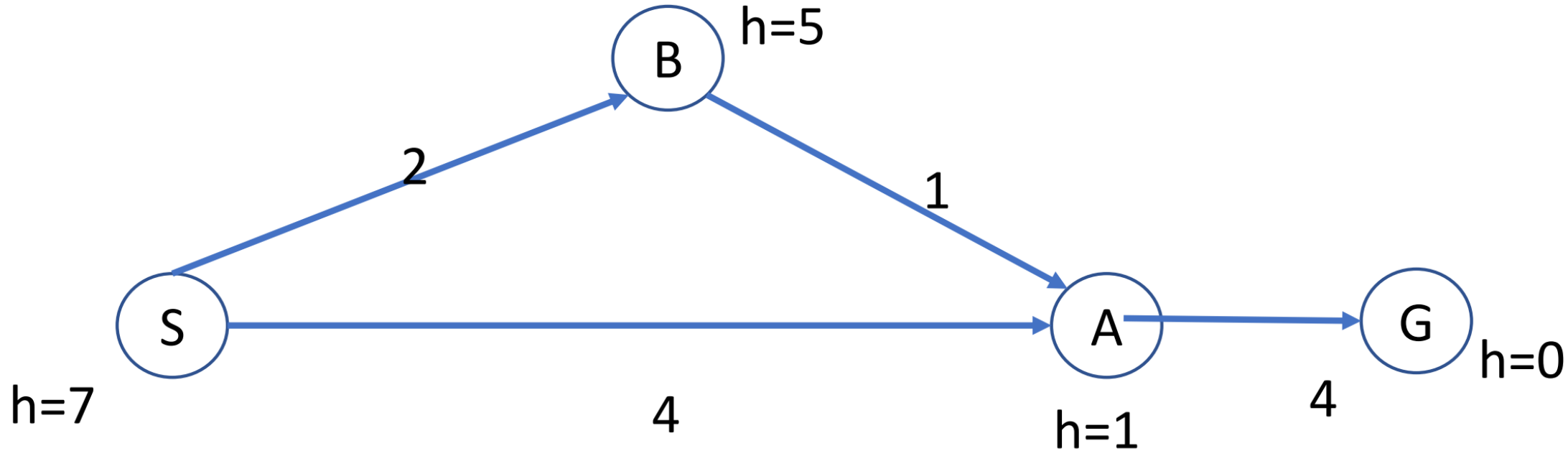
$$f(n') = g(n') + h(n')$$
$$f(n') = g(n) + c(n; a; n') + h(n')$$
$$f(n') \geq g(n) + h(n)$$
$$f(n') \geq f(n)$$

$f(n)$ - nondecreasing along any path,

monotonous

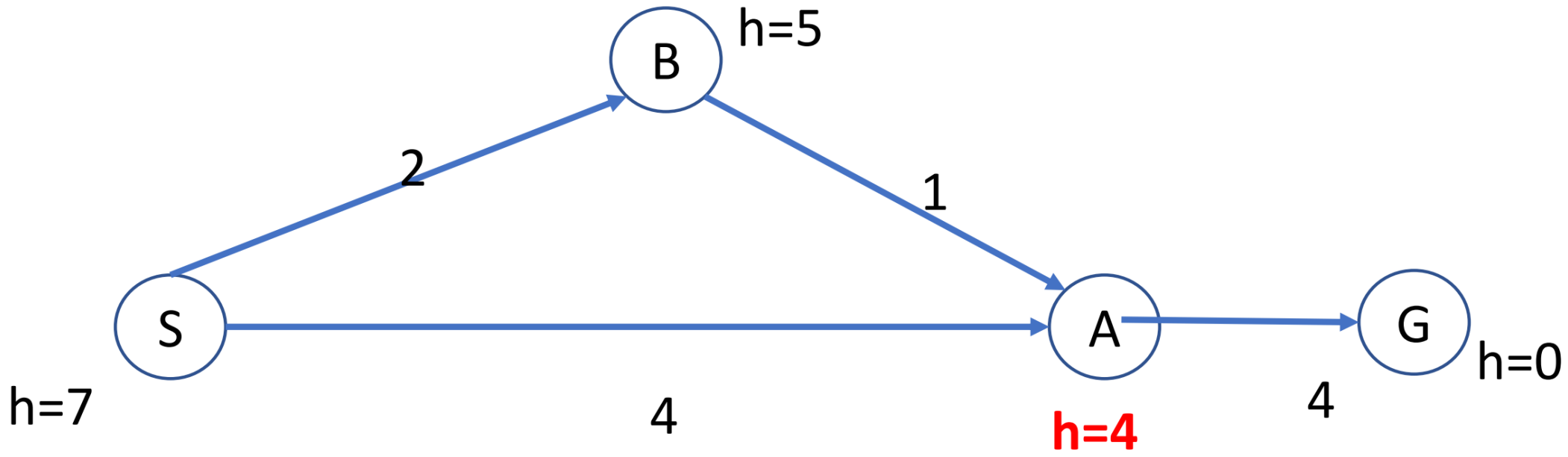


Consistency of h - Re-expansion of nodes



Consistent?

Consistency of h - Re-expansion of nodes



Proof of optimality for A*

A* with admissible heuristics cannot return a suboptimal path

Proof by contradiction:

1. Suppose C^* is the optimal cost and A^* with an **admissible** heuristic returns a suboptimal path.
2. Suppose n is a node on the optimal path.
3. If A^* returns an nonoptimal path it means that n was not expanded.
4. This means that $f(n) > C^*$ otherwise n would be expanded.
5. For admissible $h(n)$, we can derive that $f(n) \leq C^*$, which contradicts 4.

Proof of optimality for A*

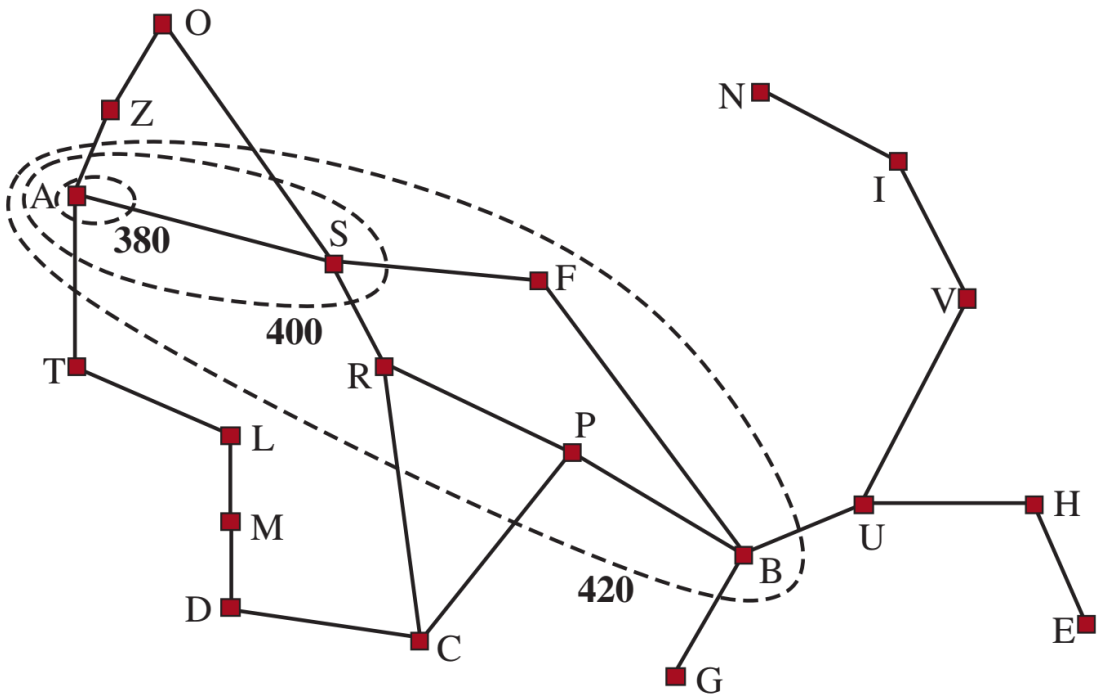
1. $f(n) = g(n) + h(n)$ by definition
2. $f(n) = g^*(n) + h(n)$ because n is on the optimal path
3. $f(n) \leq g^*(n) + h^*(n)$ because of admissibility
4. $f(n) \leq C^*$ by definition $C^* = g^*(n) + h^*(n)$

Contradicts $f(n) > C^*$

f - contours

A* expands nodes in order of increasing f.

- Surely expanded nodes
- Optimally efficient
- Pruning



Satisficing "good enough" solutions

Inadmissible heuristic

- Overestimating the optimal cost
- Risk missing the optimal solution
- Potentially can be more accurate

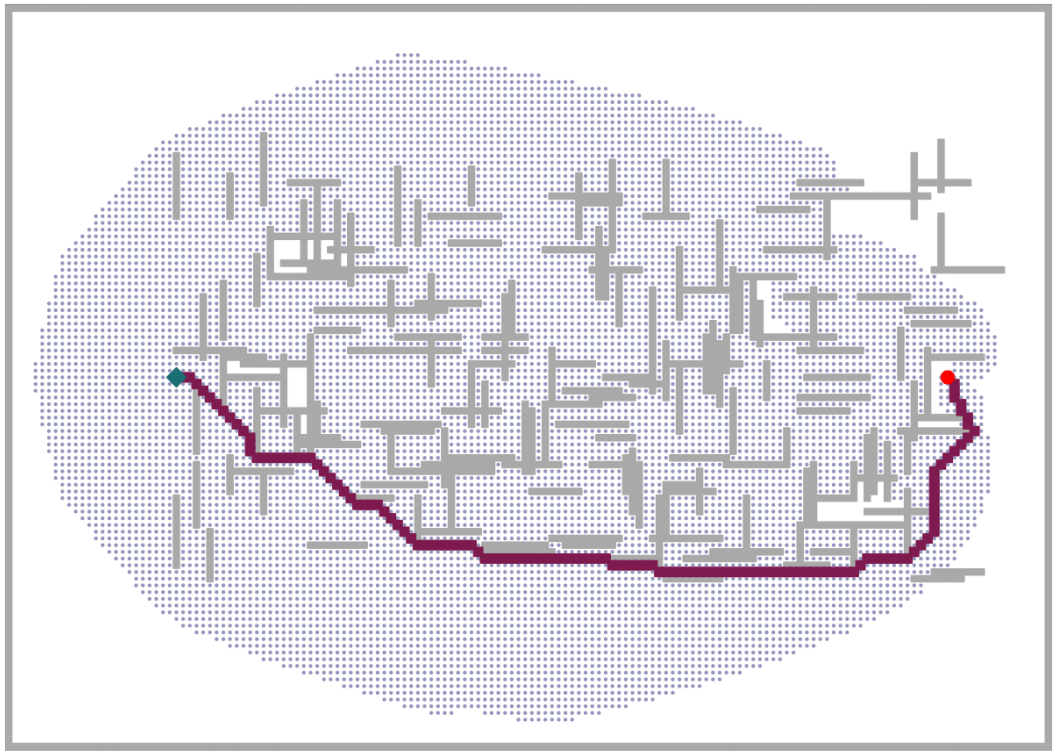
Weighted **A***:

$$f(n) = g(n) + W \times h(n),$$
$$W > 1$$

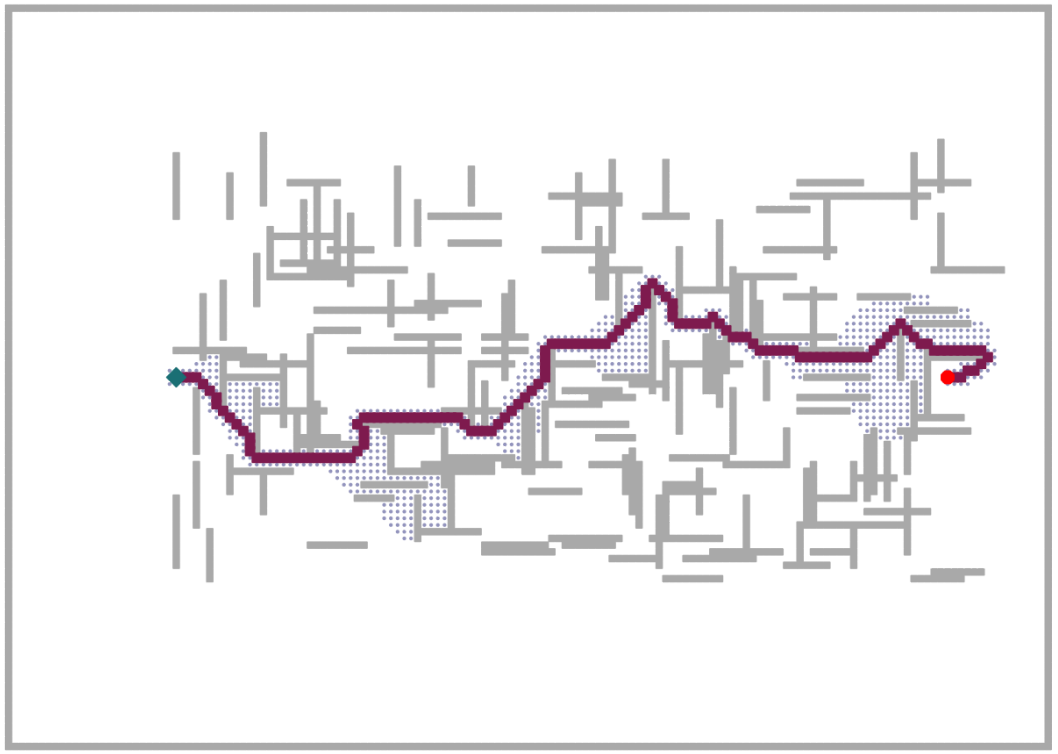
Example: *detour index*

$$1.3 * StraightLineDistance$$

Satisficing good enough solutions



(a)



(b)

(a) A* (b) Weighted A* with $W = 2$

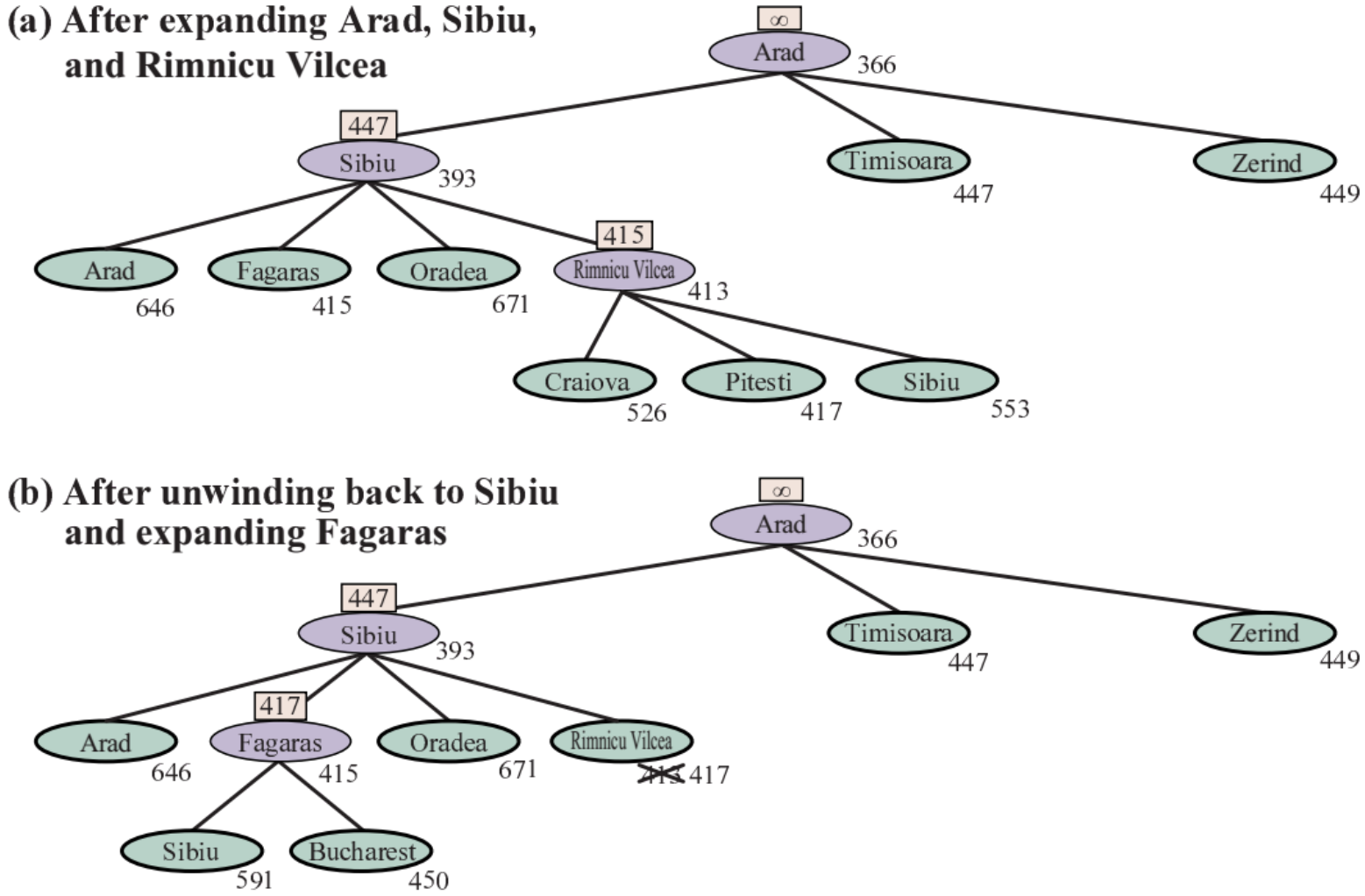
Memory-bounded search

Main issue of A* is the use of memory - keeps all nodes in memory (*reached*)

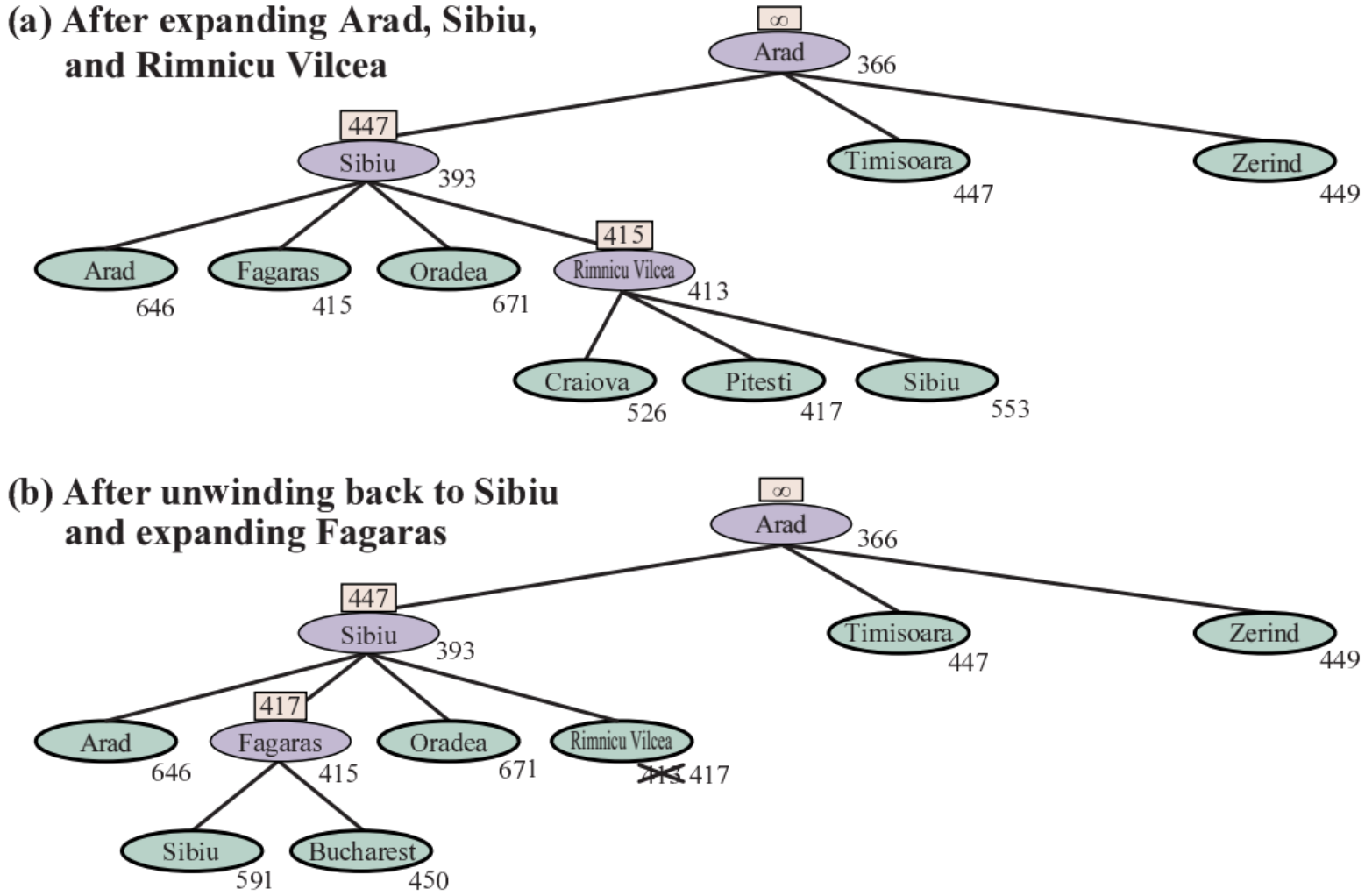
How to reduce memory use:

1. Reference count - remove a state when there are no more ways to reach it
2. Beam search - limit size of frontier to k best candidates
3. Iterative-deepening A* - gradually increase f-cost cutoff
4. Recursive best-first search (RBFS)
5. Memory-bounded A* - expand until memory is full, then drop the worst

Recursive best-first search with f-limit value



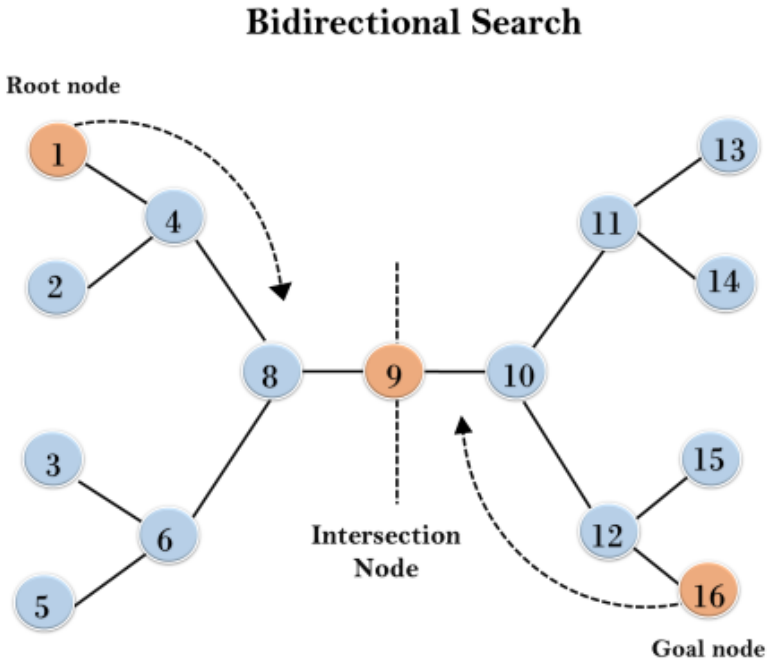
Recursive best-first search with f-limit value



Bidirectional search - Uninformed

Search forwards from initial state and backwards ffrom the goal, hoping that searches will meet.

Time and space: $O(b^{d/2}) + O(b^{d/2}) = O(b^{d/2})$, e.g. 50,000 times less than $O(b^d)$ for $b = d = 10$)



```
function BiBF-Search(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← Node(problem_F.initial)                    // Node for a start state
  node_B ← Node(problem_B.initial)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.State and value node_F
  reached_B ← a lookup table, with one key node_B.State and value node_B
  solution ← failure
  while not Terminated(solution, frontier_F, frontier_B) do
    if f_F(Top(frontier_F)) < f_B(Top(frontier_B)) then
      solution ← Proceed(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← Proceed(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function Proceed(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← Pop(frontier)
  for each child in Expand(problem, node) do
    s ← child.State
    if s not in reached or Path-Cost(child) < Path-Cost(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← Join-Nodes(dir, child, reached_2[s]))
        if Path-Cost(solution_2) < Path-Cost(solution) then
          solution ← solution_2
  return solution
```

Bidirectional search - Informed

For $f(n) = g(n) + W \times h(n)$, neither optimal nor efficient even with admissible $h(n)$.

Fix:

Ensure that we never expand node with $g(n) > \frac{C^*}{2}$.

Use $f_2(n) = max(2g(n), g(n) + h(n))$

Heuristic functions

8-puzzle - $9!/2 = 181000$ states

15-puzzle - $16!/2 = 10000000000000$ states

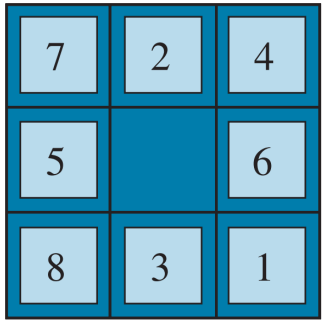
$h_1(n)$ = number of misplaced tiles,

e.g. 8

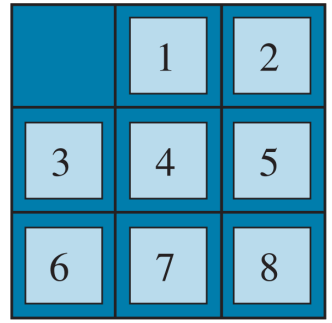
$h_2(n)$ = **Manhattan distance** - number of squares from desired location of each tile,

e.g. 3 + 1 + 2 + 2 + 3 + 2 + 2 + 3 = 18

Shortest solution cost is 26 actions.



Start State

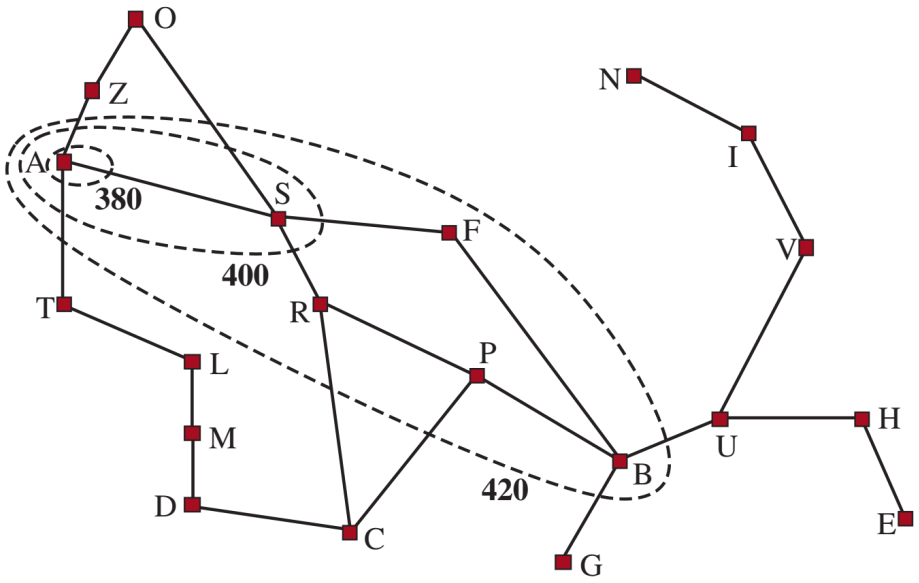


Goal State

Effective branching factor for h

Braching factor that a uniform tree of depth d would have to contain $N + 1$ nodes.

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)d$$



	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Heuristic dominance

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

Given any admissible heuristics h_a, h_b,
$h(n) = max(h_a(n); h_b(n))$
is also admissible and dominates h_a, h_b

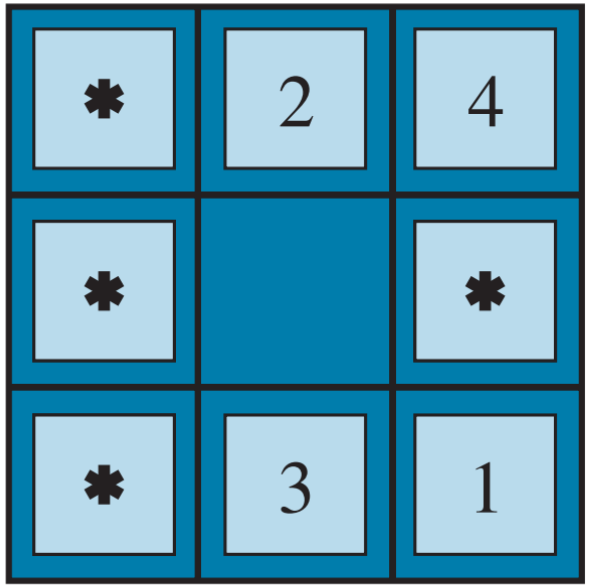
Generating heuristics from relaxed problems

Relaxed problem - simplified version of the problem, fewer restrictions, with shortcuts.

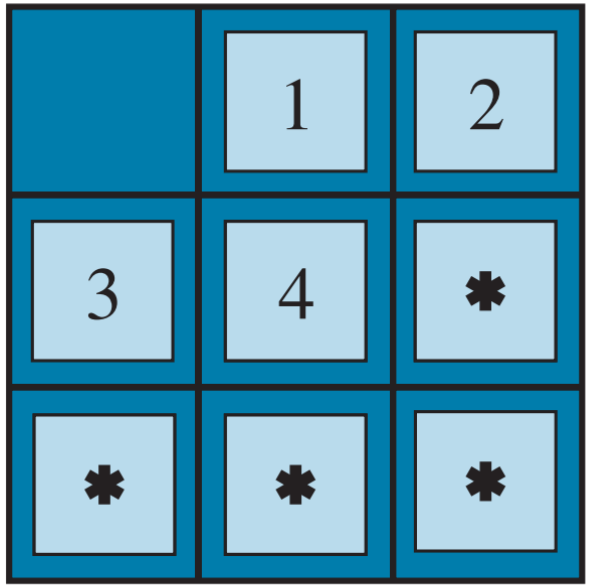
Examples for 8-puzzle:

a. A tile can move from square X to square Y if X is adjacent to Y.
b. A tile can move from square X to square Y if Y is blank.
c. A tile can move from square X to square Y.

Generating heuristics from subproblems
Pattern databases



Start State



Goal State

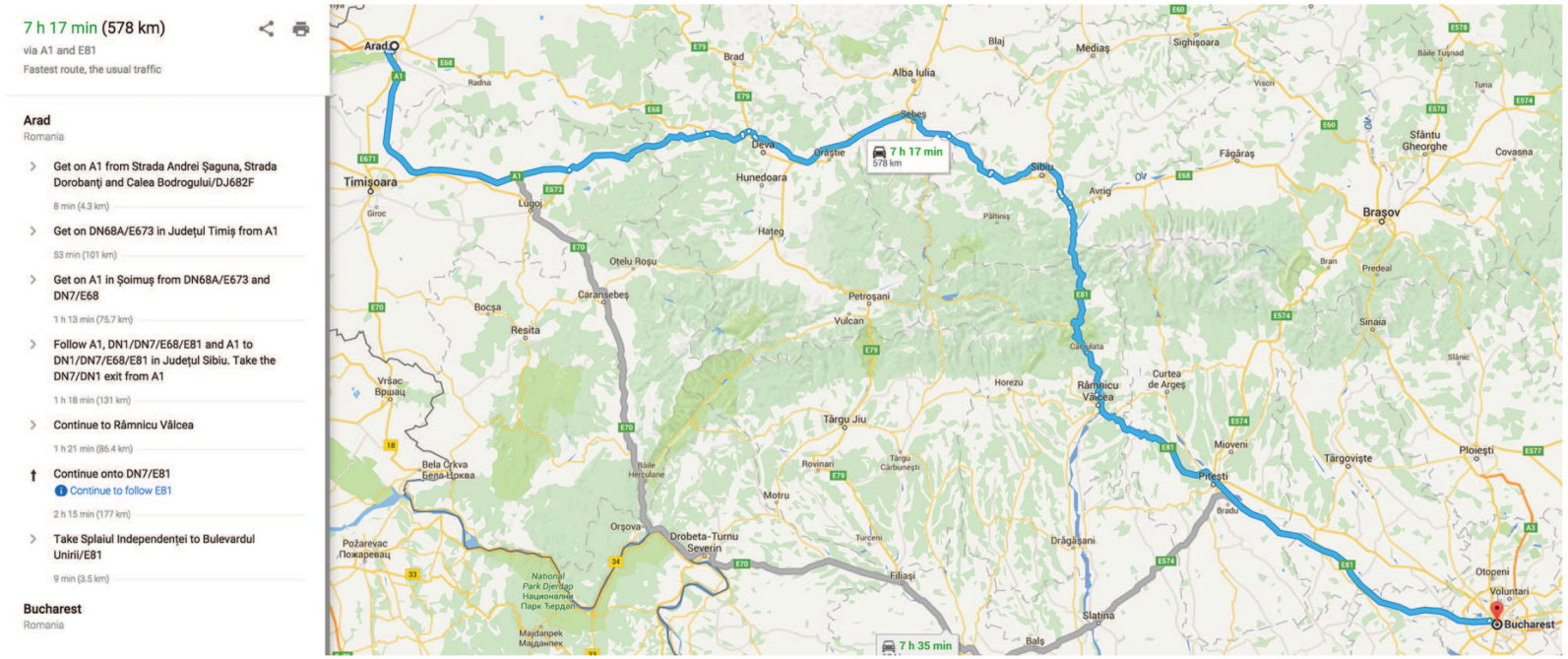
9 × 8 × 7 × 6 × 5 = 15,120 patterns in the database

Generating heuristics with landmarks

Precomputation of some optimal path costs.

$$h_L(n) = min_{L \in Landmarks} C^*(n, L) + C^*(L, goal)$$

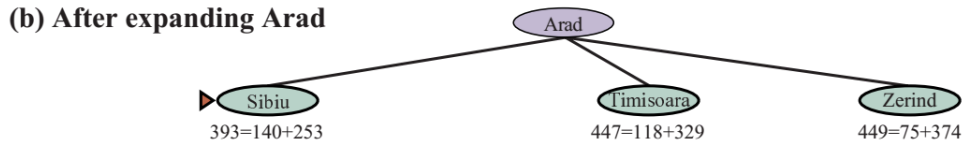
Admissible?

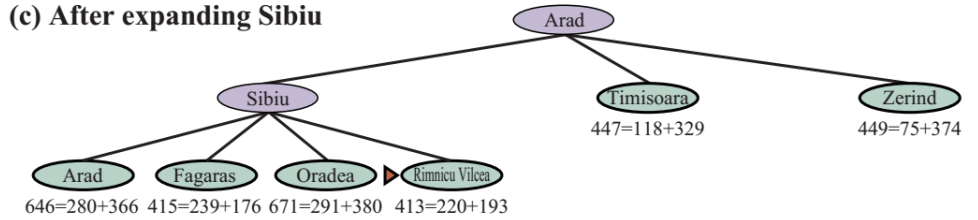


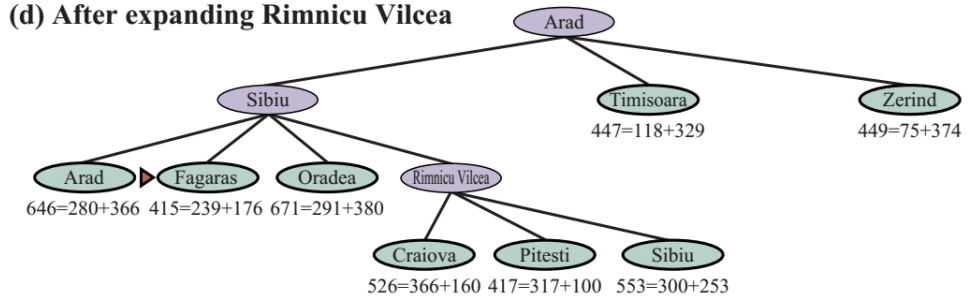
Learning to search

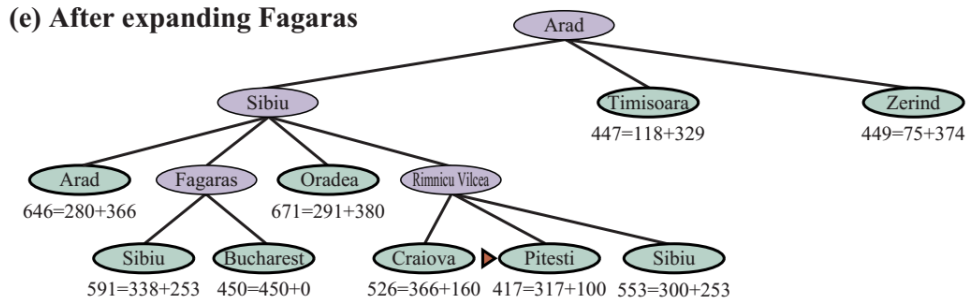
Metalevel state space - internal state of a program,
e.g. gradually expanding tress

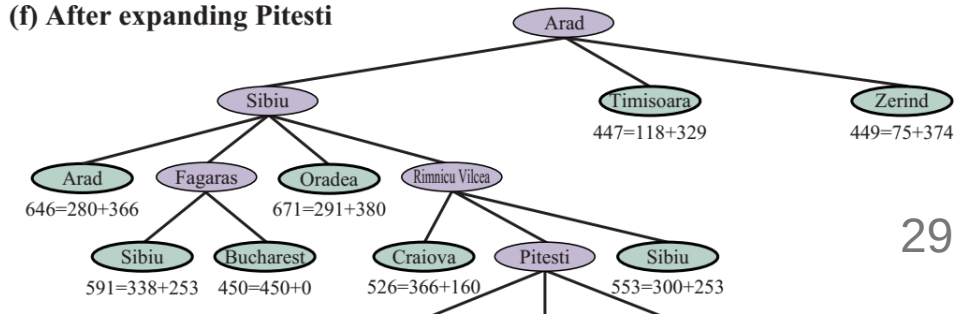
Learning heuristics from experience:
$h(n) = c_1 x_1(n) + c_2 x_2(n)$











Search in complex environments

1. Local search
    - Hill Climbing
    - Simulated annealing
    - Local beam
    - Genetic algorithms
2. Search in non-deterministic environments
3. Search in partially-observable environments

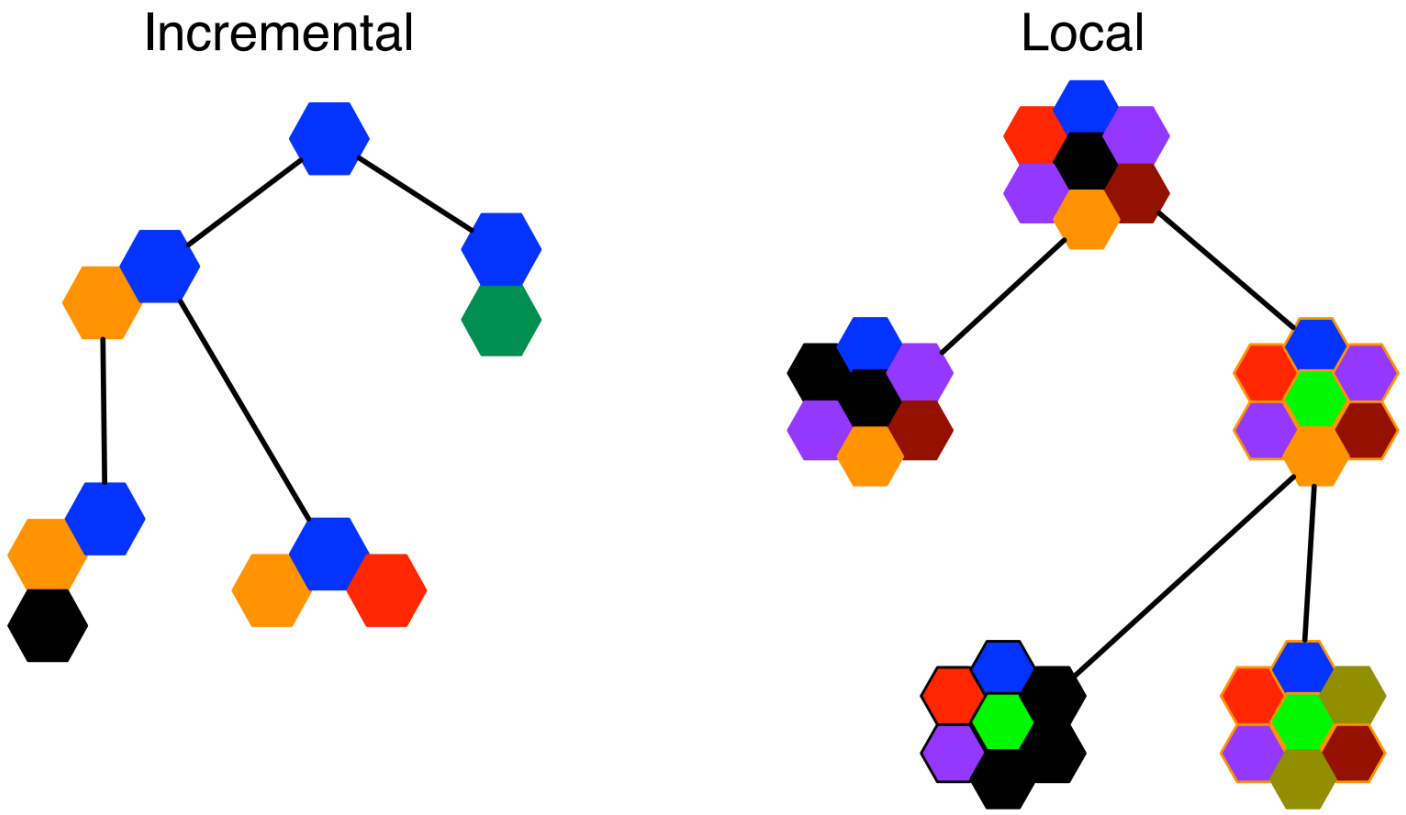
Local search

Path is irrelevant, **goal state** is the solution.

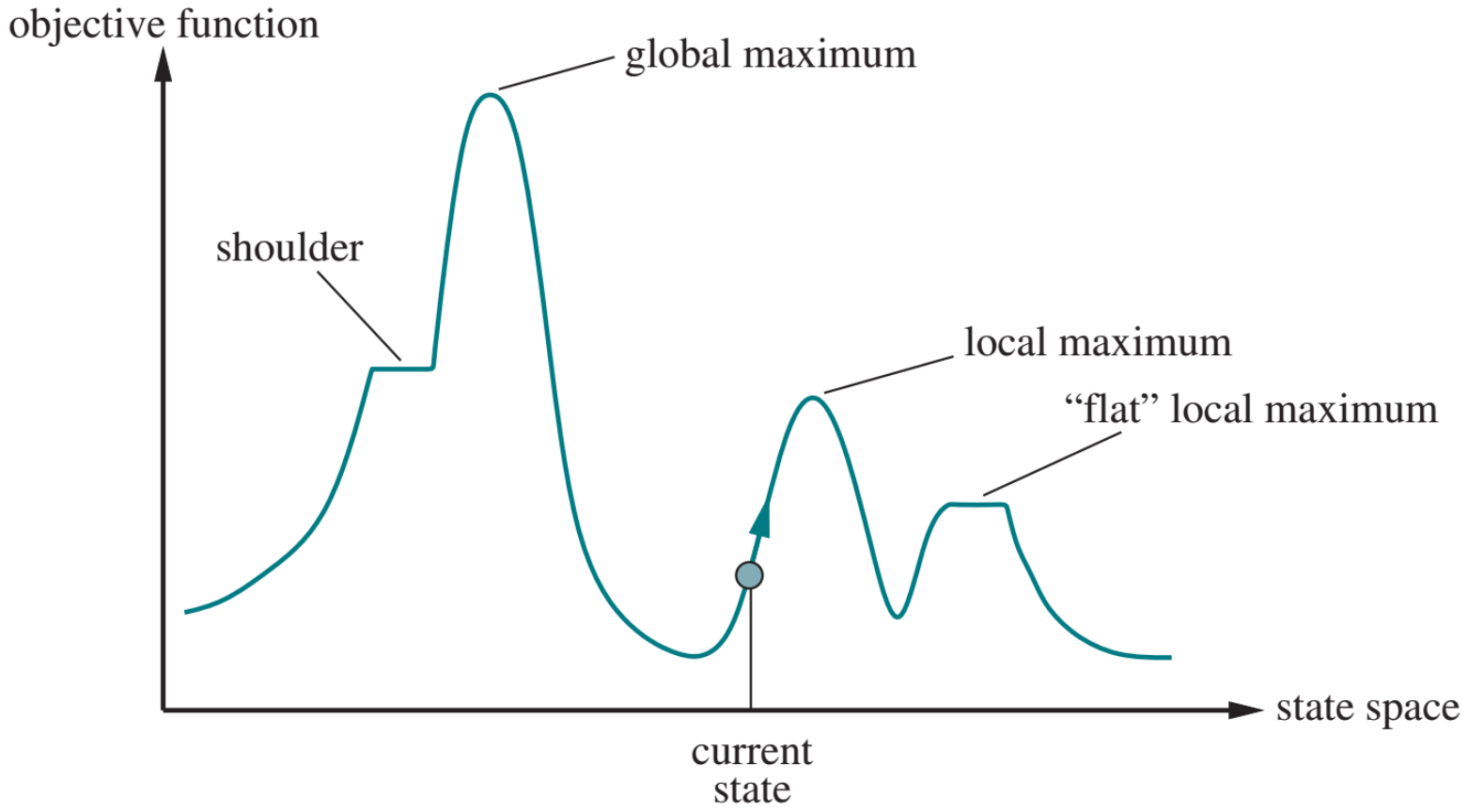
State space = set of "complete" configurations.
e.g. find optimal layout or timetable that satisifes constraints

Idea: Search from current state(s) to neighbour states to find a better one.

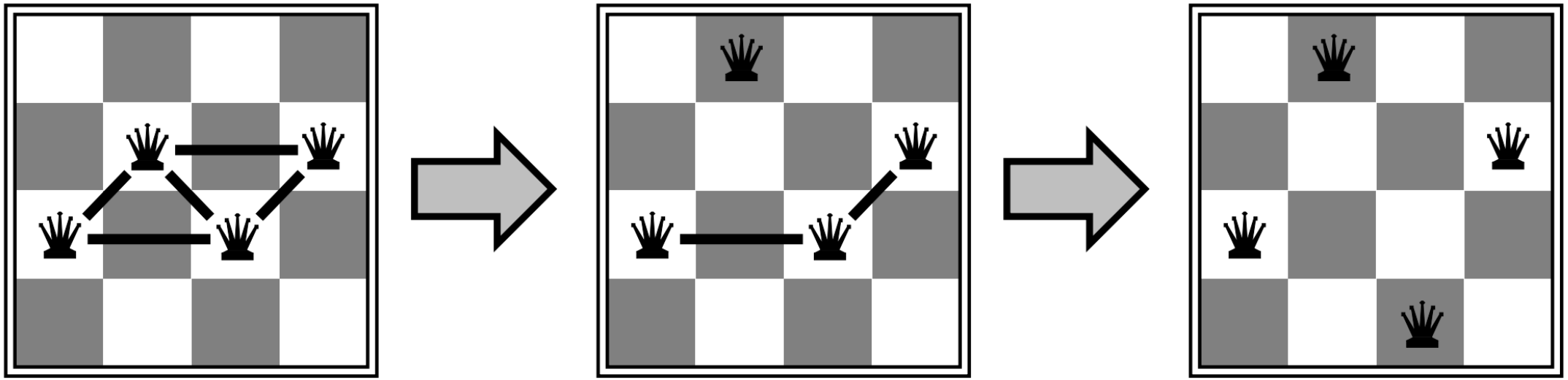
Incremental vs local search



State-space landscape



Example: n-queens

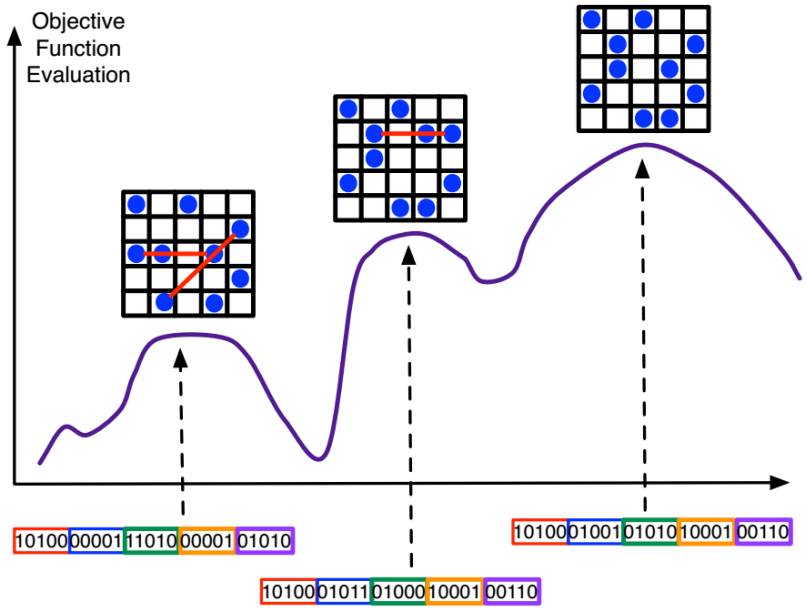


Properties of local search

- Uses very **little memory** - only keeps one (or a set) of current states, not paths or reached states.
- Often finds **reasonable solutions** in large or infinite state space
- Can solve **optimization problems**, finding the best state according to an **objective function**

Hill-climbing search

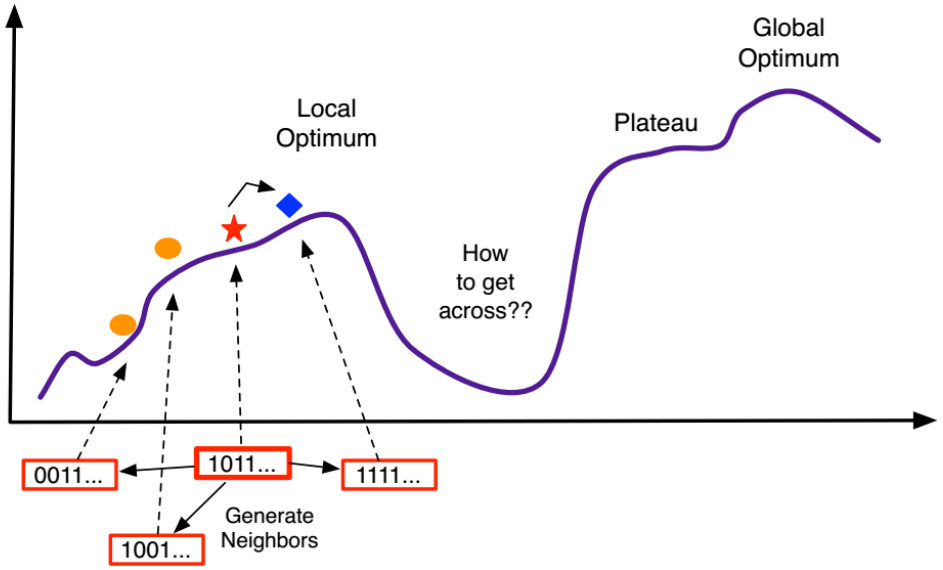
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current*
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    current ← *neighbor*



Hill-climbing gets stuck

- Local maxima
- Ridges
- Plateaus

"Like climbing Everest
in thick fog with amnesia"



Hot to not get stuck

- Sideways move
- Sochastic hill climbing
- First-choice hill climbing
- Random-restart hill climbing

Simulated annealing

Idea: Escape local minima by allowing some "bad" moves.

Gradually decrease their size and frequency.

function SIMULATED-ANNEALING($problem, schedule$) **returns** a solution state
  $current \leftarrow problem.$INITIAL
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** $current$
    $next \leftarrow$ a randomly selected successor of $current$
    $\Delta E \leftarrow$ VALUE($current$) – VALUE($next$)
    if $\Delta E > 0$ **then** $current \leftarrow next$
    else $current \leftarrow next$ only with probability $e^{-\Delta E/T}$

Local beam search

Idea: Keep k states instead of 1; choose top k of all their successors.

- Not the same as k searches run in parallel.
- Searches that find good states "recruit" other searches to join them.

Problem: Ofte all k states end up on same local hill.

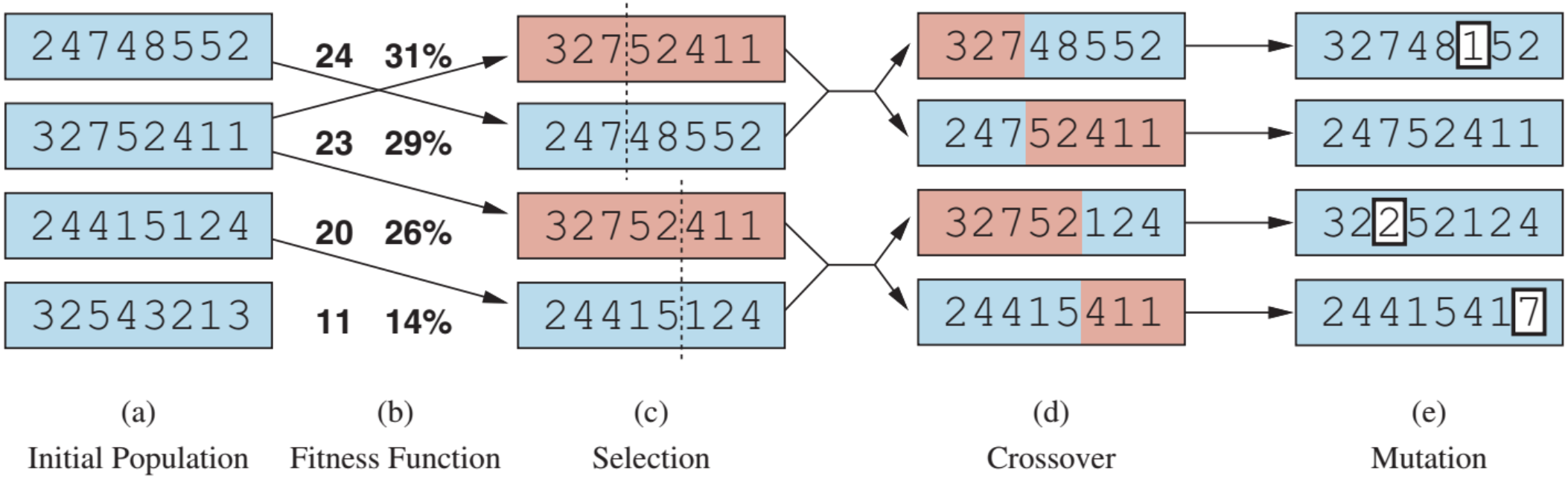
Idea: Choose k successors randomly, biased towards good ones.

Analogy to natural selection!

Genetic algorithms

1. Start with a **population** of *k* randomly generated states
2. Randomly choose two **parent** states weighted by their **fitness**
3. Generate a **child** state by combining **parent** states randomly.
4. **Mutate** the **child** state by introducing random changes.
5. Add **child** to the **population**.
6. Repeat from 2 until **child** is fit enough or time has elapsed.

Genetic algorithms

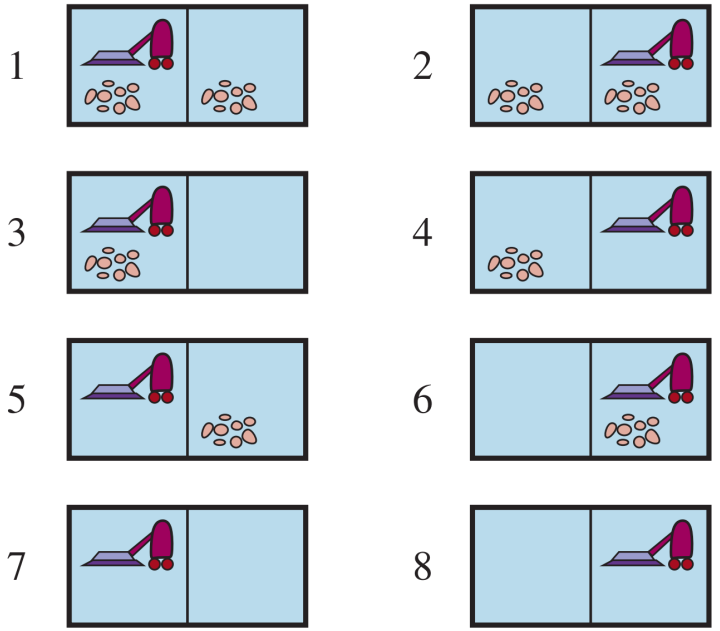


Search with nondeterministic actions

Example: Erratic vacuum world

Non-deterministic suck action:

- When applied to a dirty square, it cleans the square and sometimes cleans and adjacent square too.
  e.g. **Results(1, Suck) = {5,7}**
- When applied to a clean square, it may deposit dirt on the square.
  e.g. **Results(7, Suck) = {7,3}**



Solution in state 1:

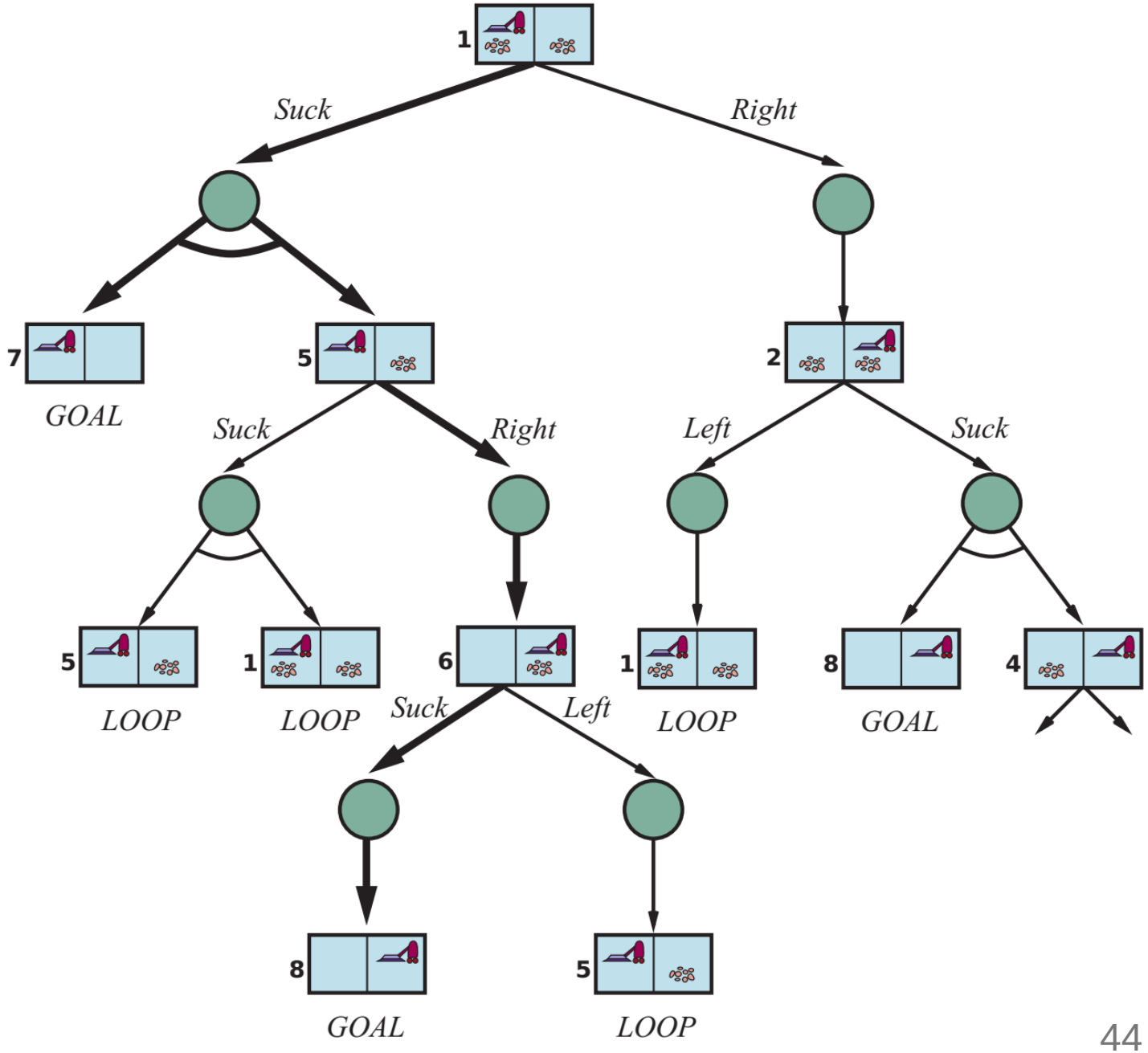
[Suck, if State = 5 then [Right, Suck] else []]

AND-OR search trees

OR nodes - actions

AND nodes - outcomes

(**belief states**)



Recusive, depth-first and-or graph search

```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
    return OR-SEARCH(problem, problem.INITIAL, [ ])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
    if problem.IS-GOAL(state) then return the empty plan
    if IS-CYCLE(path) then return failure
    for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
        if plan ≠ failure then return [action] + plan]
    return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
    for each s_i in states do
        plan_i ← OR-SEARCH(problem, s_i, path)
        if plan_i = failure then return failure
    return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n−1} then plan_{n−1} else plan_n]
```

Cyclic solutions

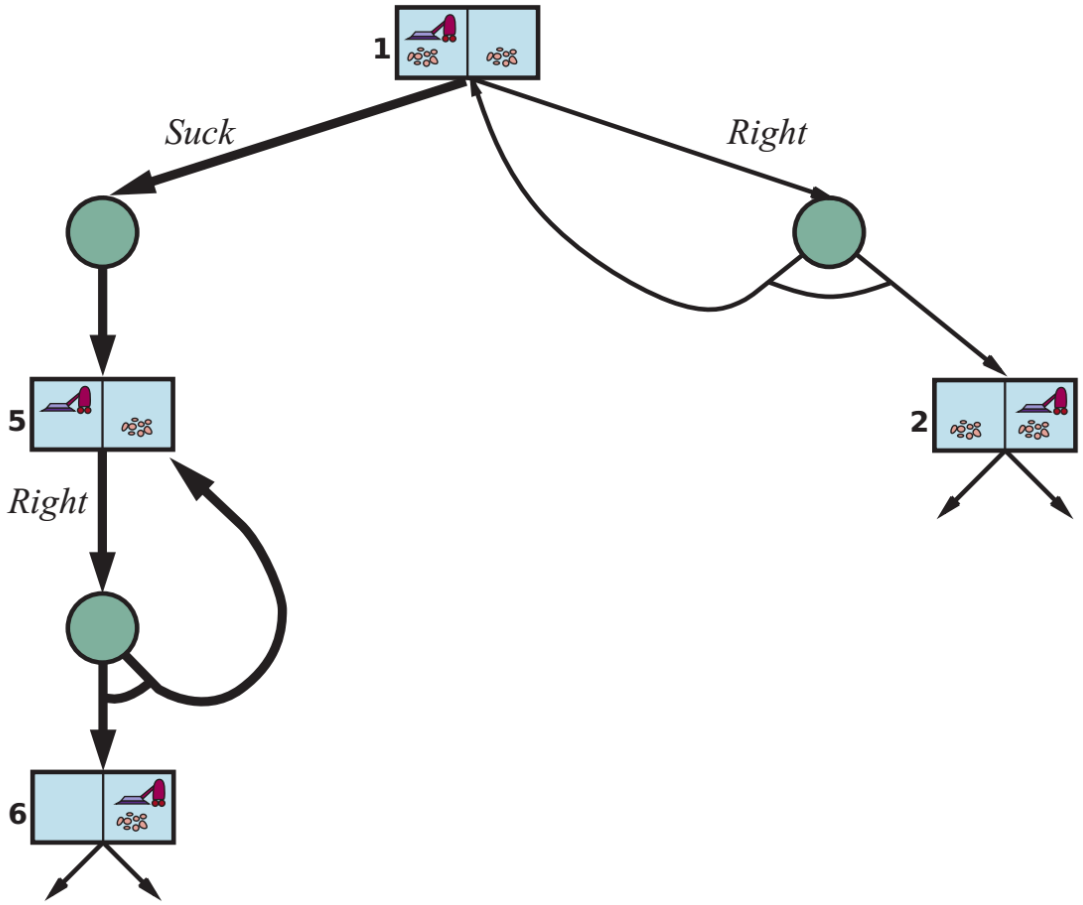
Example: Slippery vacuum world

Movement actions sometimes fail.

e.g. **Results(1, Right) = {1,2}**

Cyclic solution:

[Suck, while State = 5 do Right, Suck]

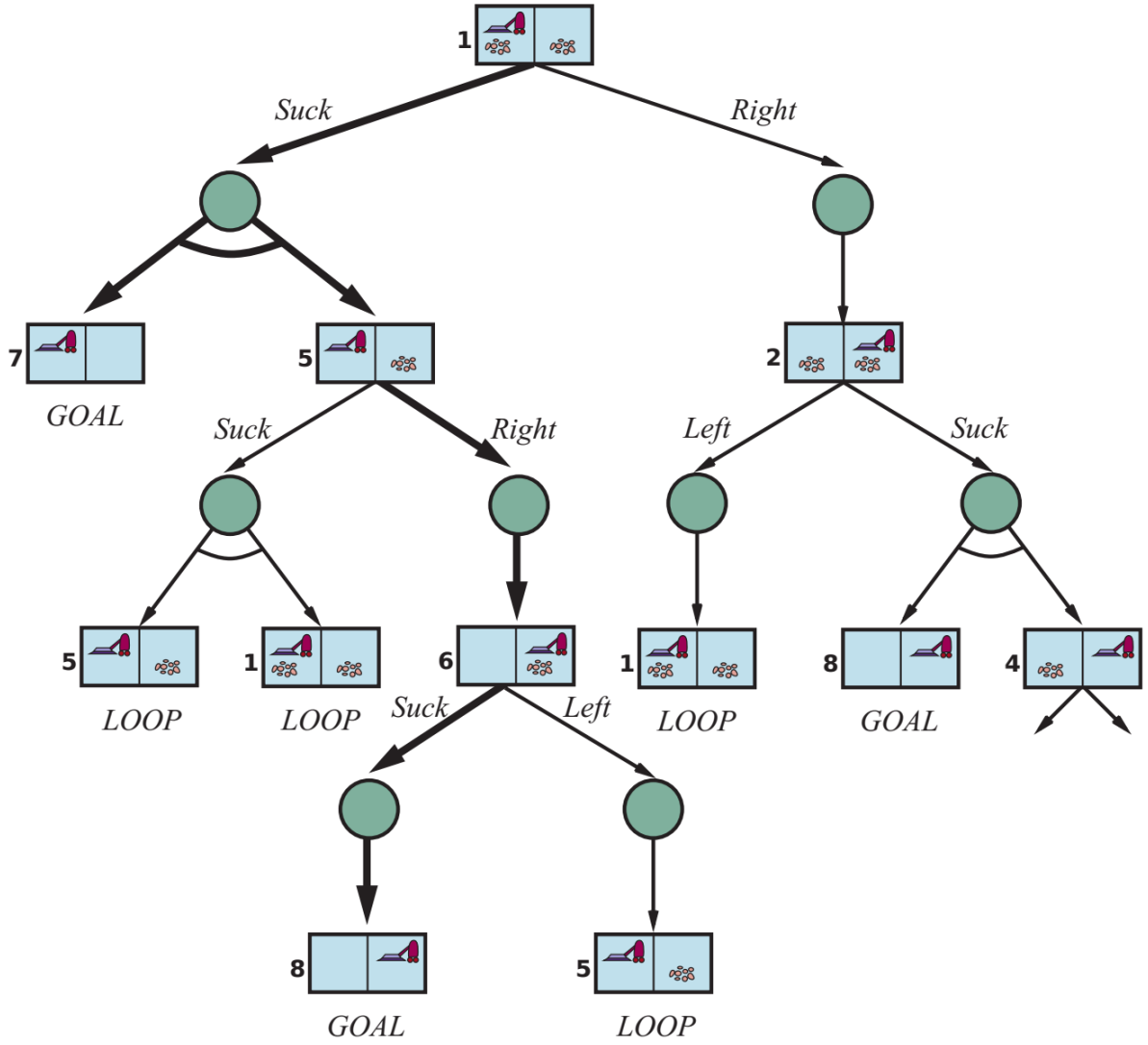


Search in partially observable environments

Partial or noisy observations are not enough to pin down the exact state.

Sensorless problem - no information from percepts at all.

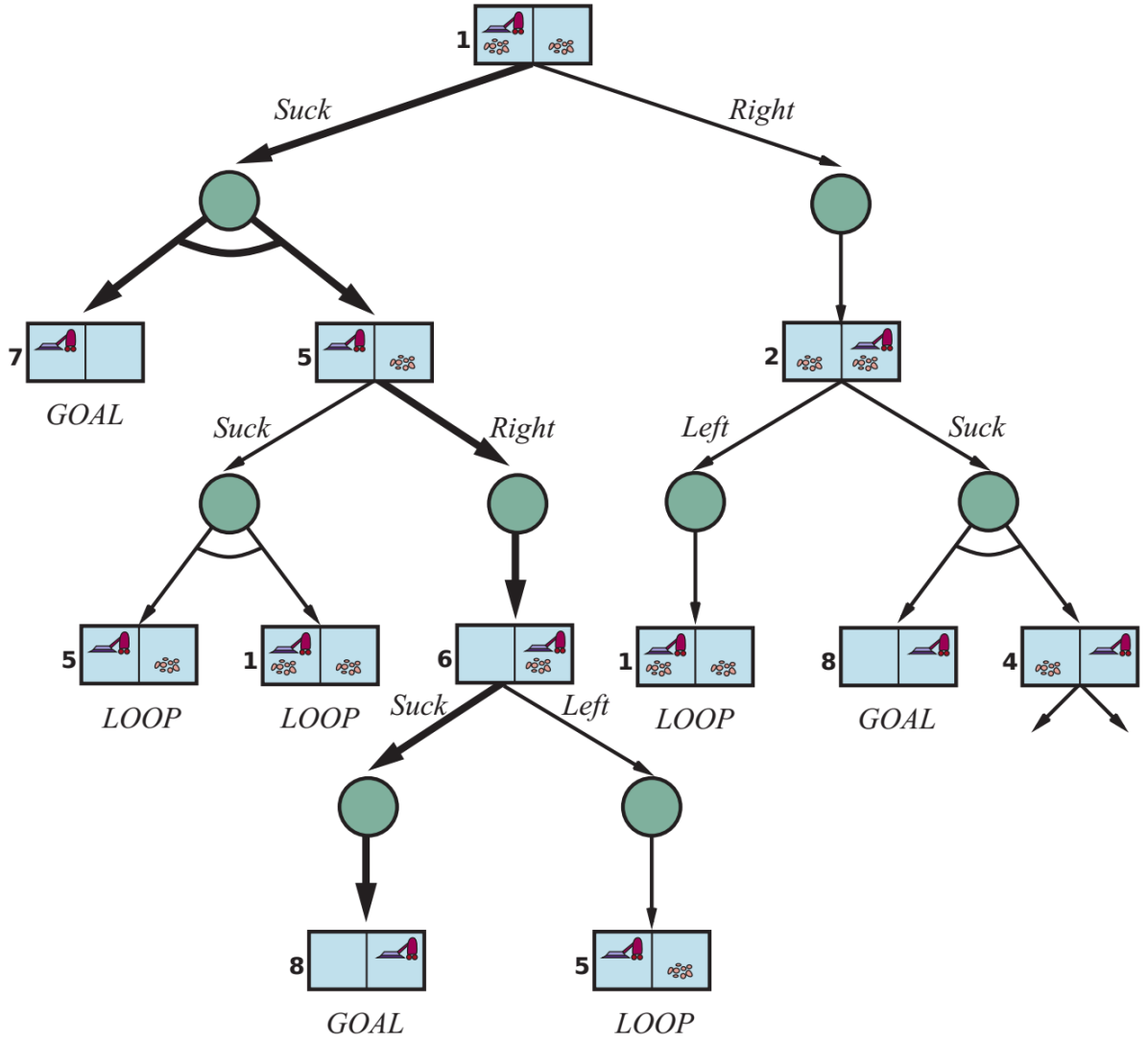
How to solve?



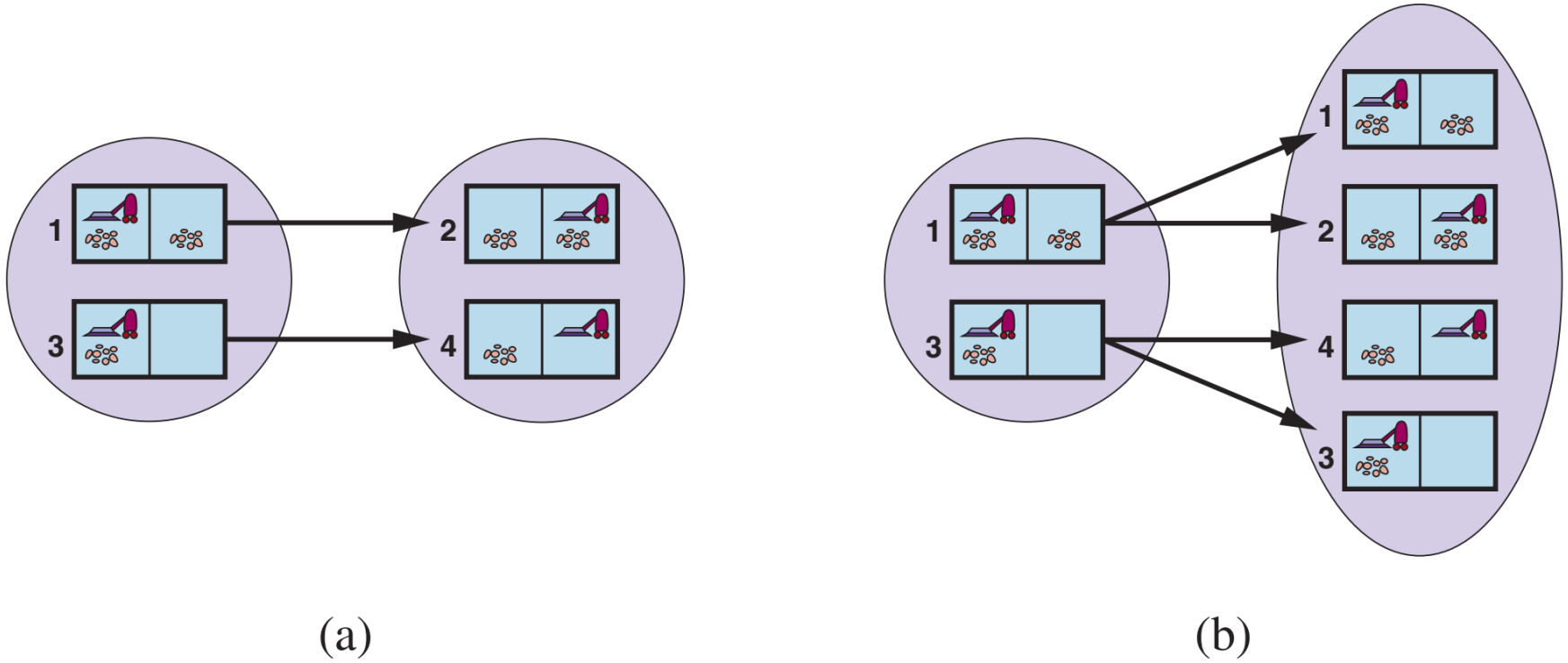
Sensorless (deterministic) vacuum world

Initial belief state: {1,2,3,4,5,6,7,8}

1. Result({1,2,3,4,5,6,7,8}, Right) = {2,4,6,8}
2. Result({2,4,6,8}, Suck) = {4,8}
3. Result({4,8}, Left) = {1,7}
4. Result({1,7}, Suck) = {7}



Predicting the next belief state with sensorless agents



(a) deterministic *Right* action (b) nondeterministic (slippery) *Right* action

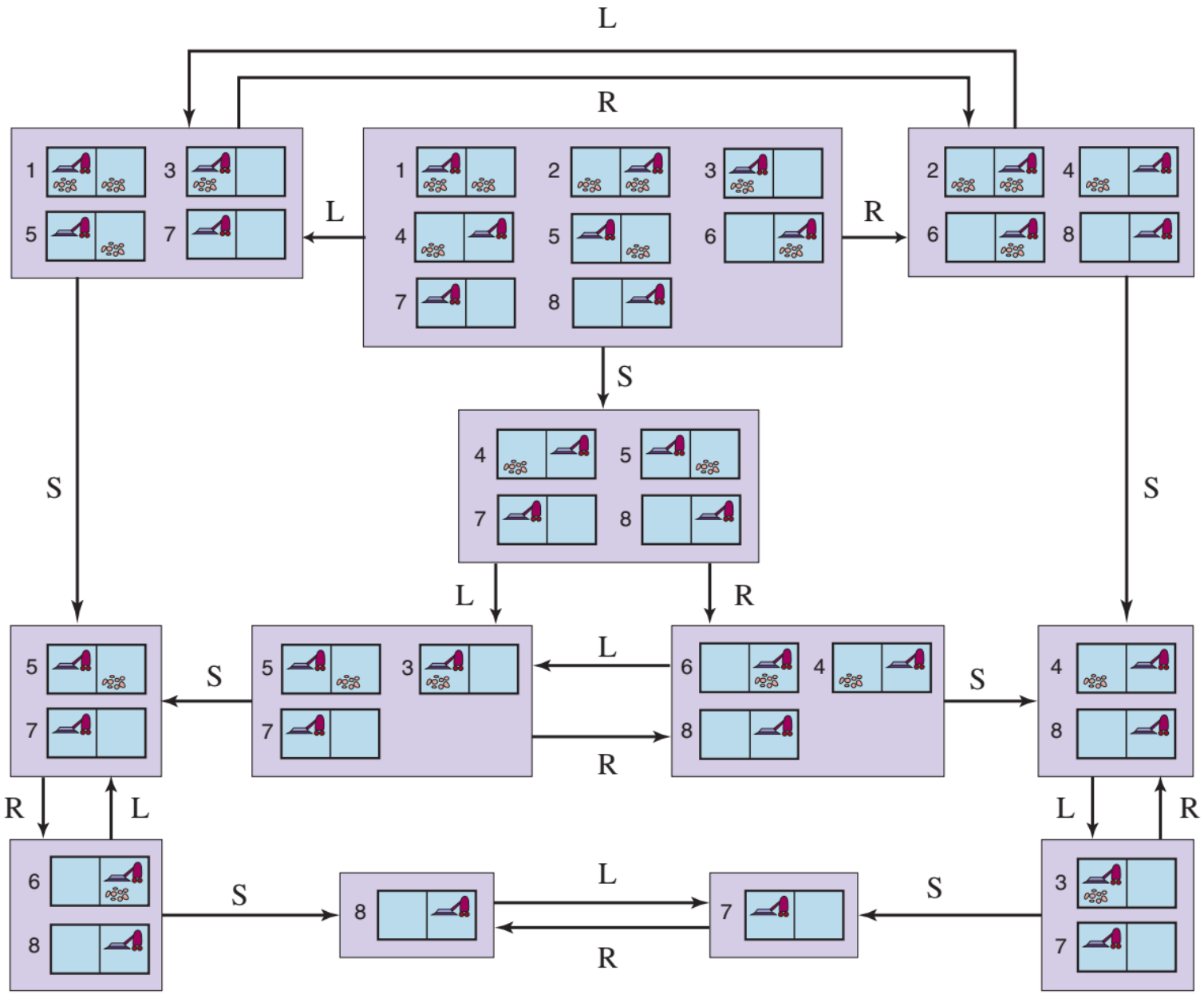
Sensorless belief-state space

Deterministic vacuum world

Possibly vs necessarily achieve goal

Prune supersets of reached belief state.

Problem: 2^N search space

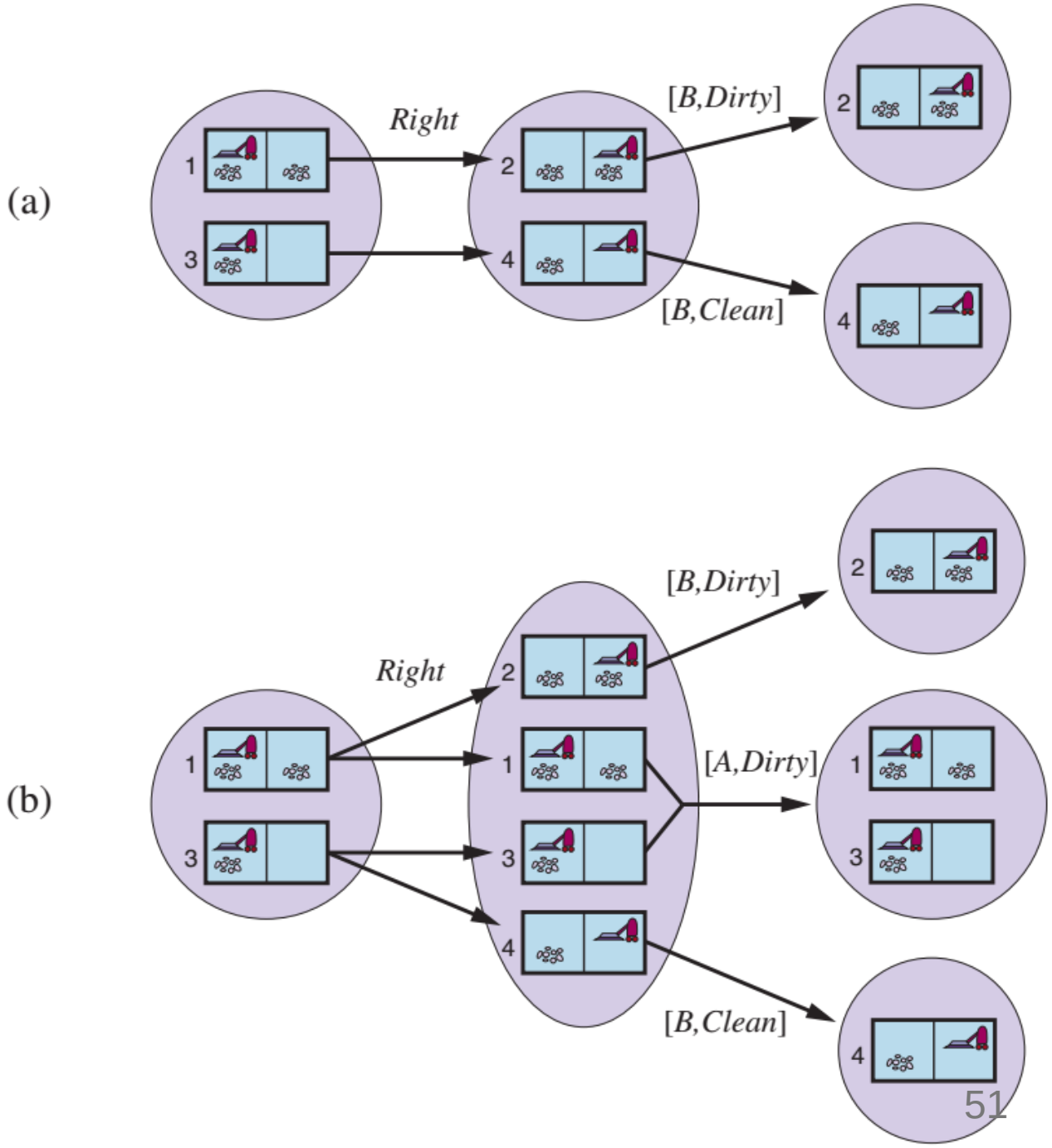


Transition model local sensing agent

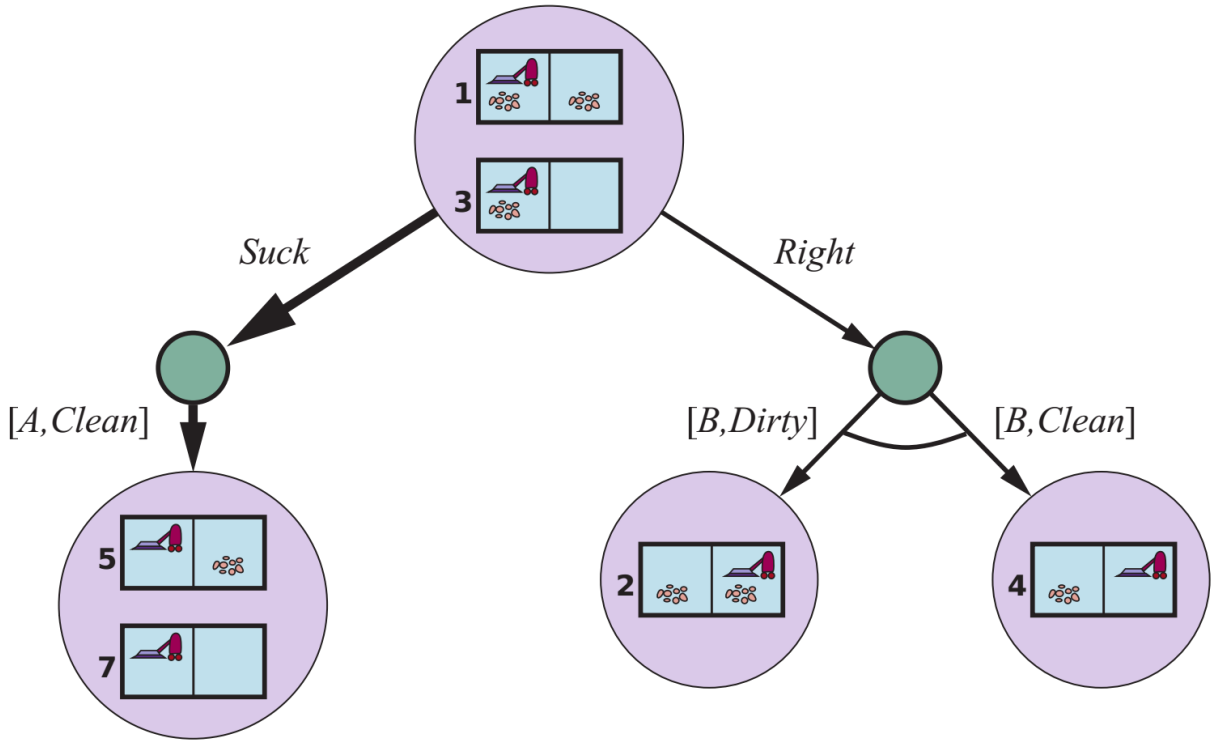
1. $\hat{b}$ = Predict(b, a)
2. o = PossiblePercepts($\hat{b}$)
3. b_o = Update($\hat{b}$, o)

(a) deterministic

(b) nondeterministic (slippery)



AND-OR search for local sensing agent



Questions?