TDT4136 Introduction to Artificial Intelligence

Lecture 4: (A* + Search in Complex Environments)

Chapter (3/)4 in the textbook.

Pinar Ozturk

Norwegian University of Science and Technology
2023

Outline

- A* Search
- Local search algorithms
  - incremental vs iterative search
  - Hill Climbing, Simulated annealing, local beam, genetic algorithms
- Search in non-deterministic environments
- Search in partially-observable environments

Uninformed Search disadvantageous

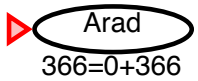
- Last week: Uninformed search
- Uninformed search, systematically searching the search space blindly - not questioning where the goal may be in the space, and not using any domain-specific knowledge
- Search space is often very large. Time/space problems with such exhaustive search - think of chess.
- In such situations, better to use algorithms which does a more informed search

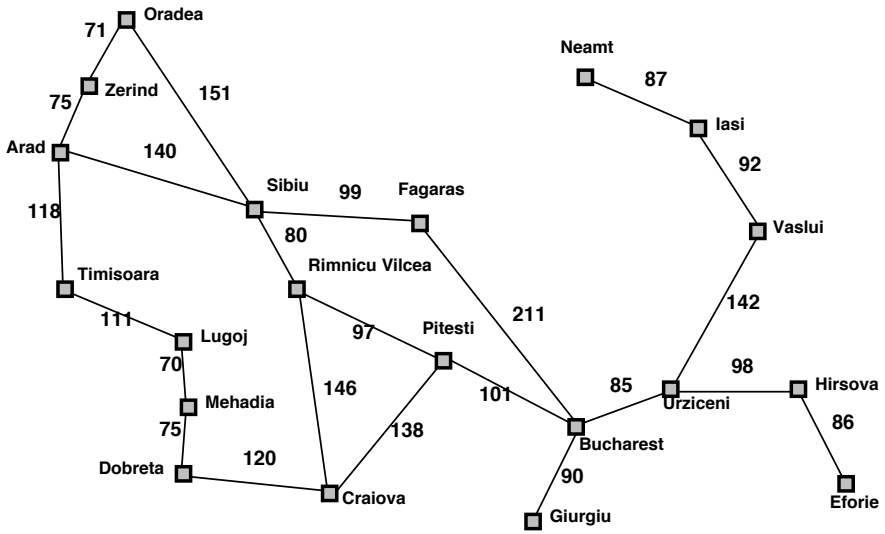
A* search

Evaluation function $f(n) = g(n) + h(n)$

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

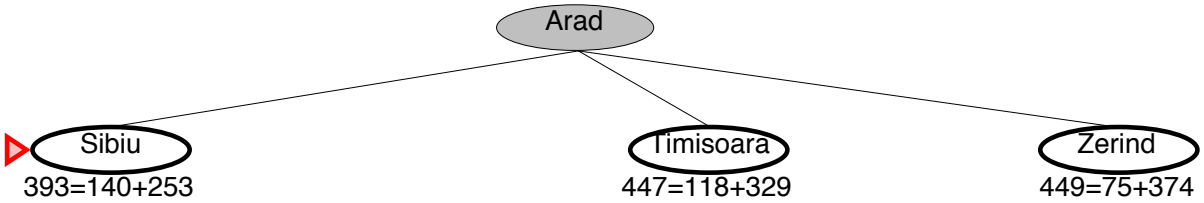
A* search example

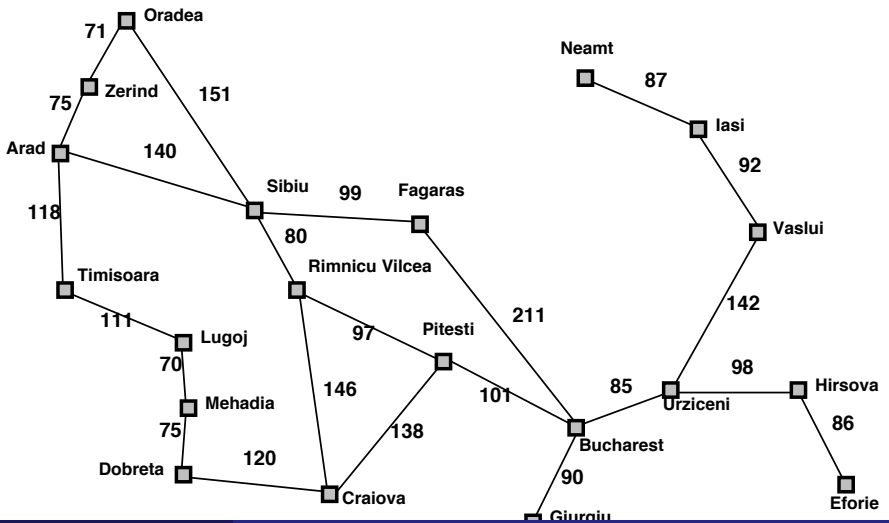




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

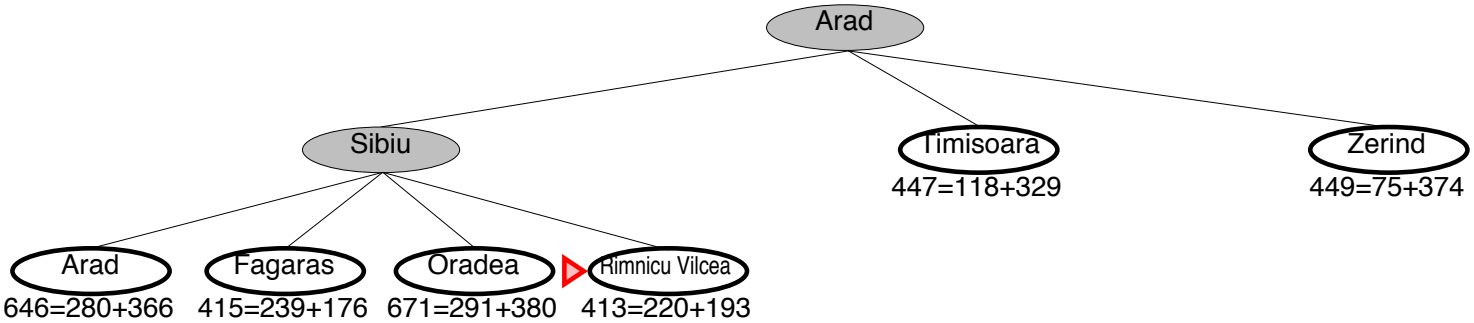


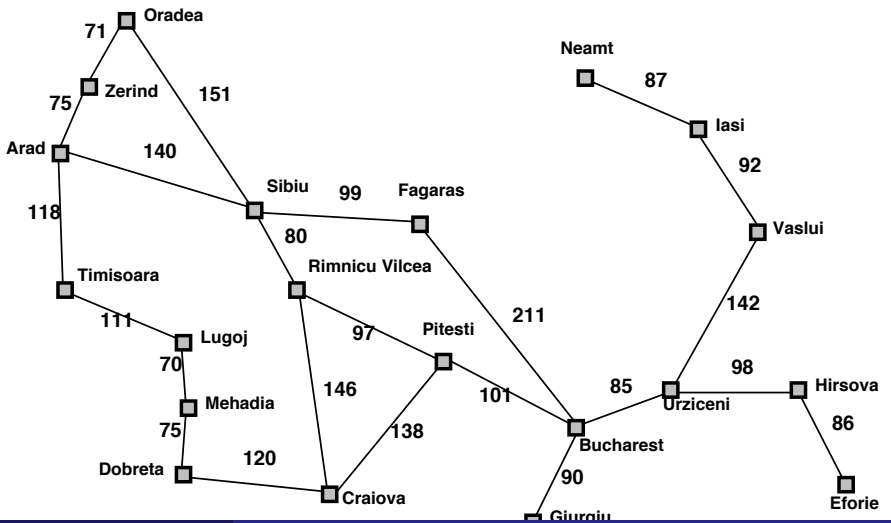


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	

A* search example

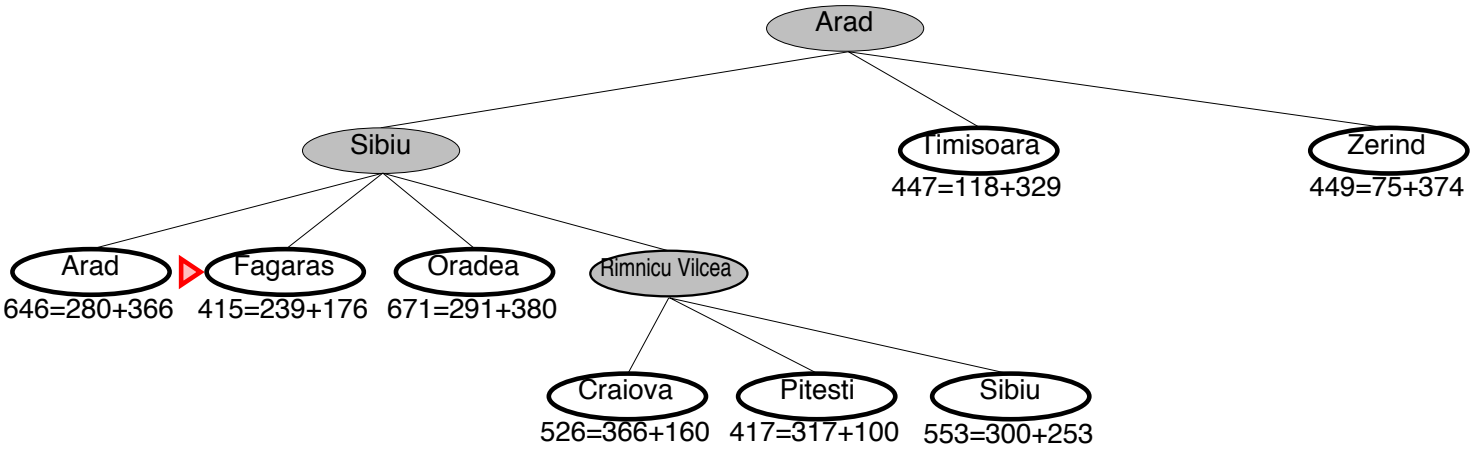




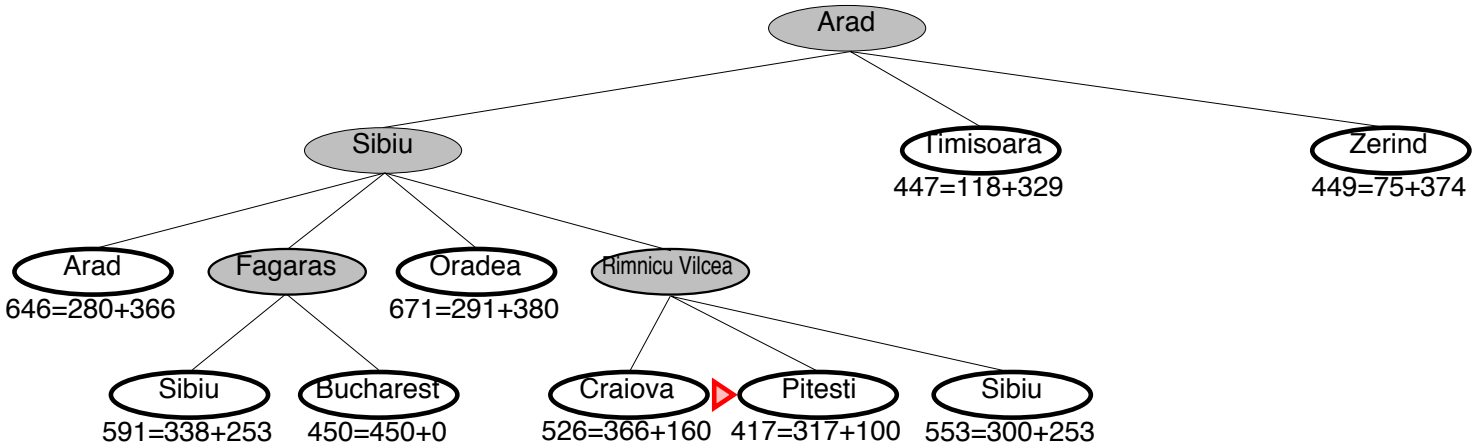
Straight–line distance to Bucharest

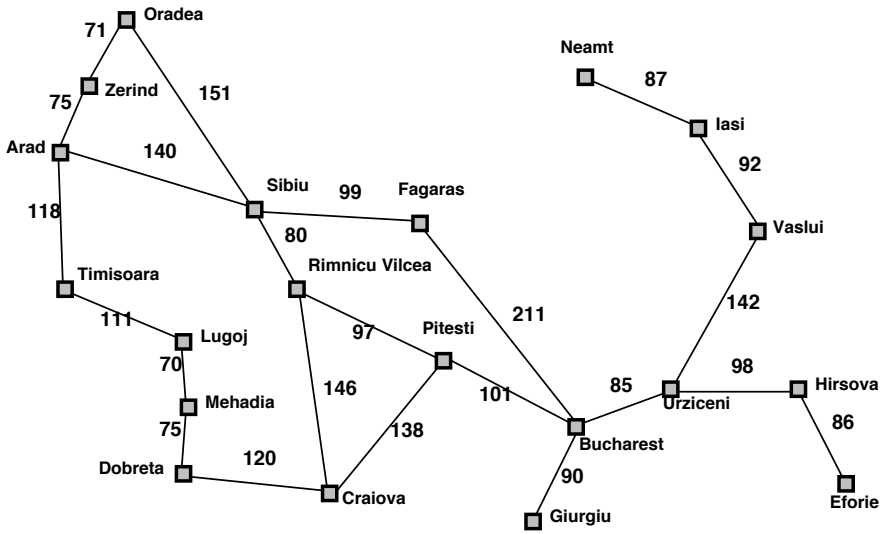
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	

A* search example



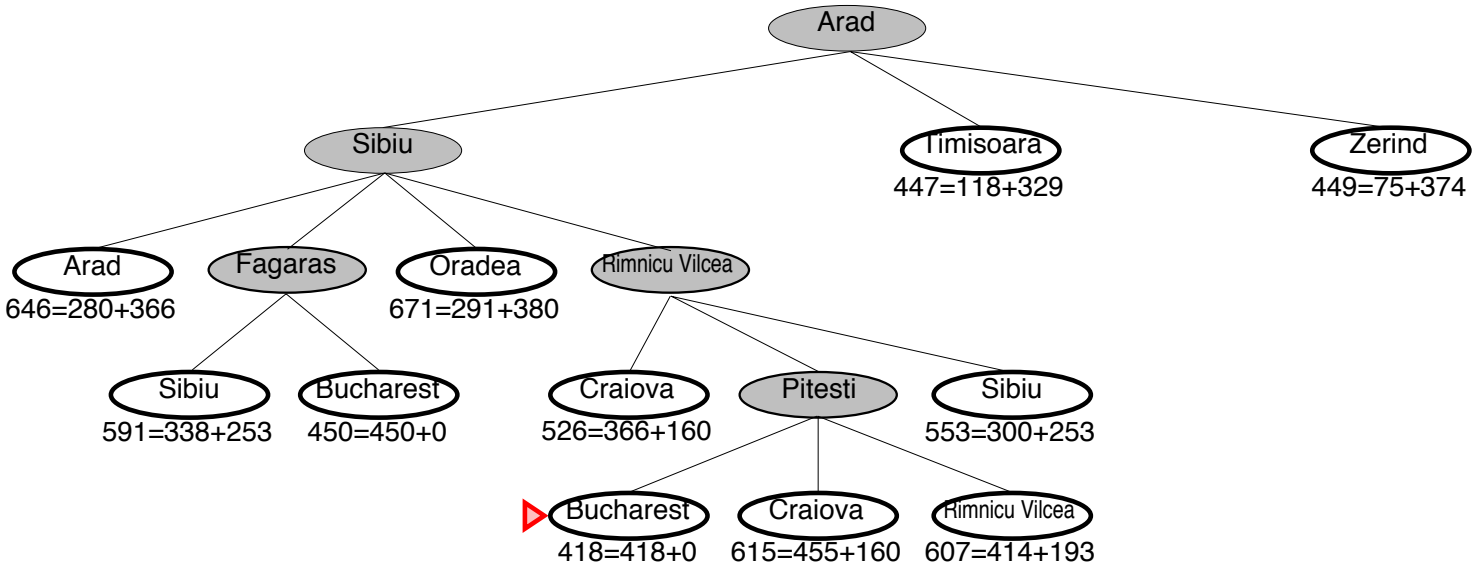
A* search example

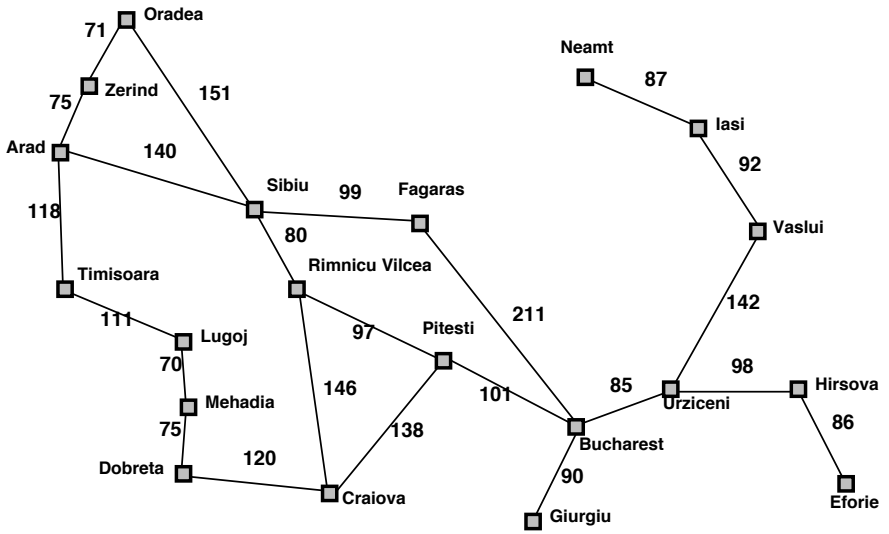




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example





Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Properties of A*

- Complete: Yes, for positive action costs, state space has a solution or is finite.
- Cost optimal: Yes, when
  - heuristic is **admissible** and non-negative
  - arc costs are positive

Properties of A*

- Time and Space complexity:
  - $O(b^{*d})$, where b^* is effective branching factor.
  - for good heuristic $b^* < b$
  - complexity increases with the error in heuristic
- Optimally efficient: Yes, if
  - heuristic is **consistent** and non-negative
    - doesn't need to re-expanded nodes

Admissibility of h

- A* search is cost optimal with an admissible heuristic.
- The heuristic is **admissible** if it never overestimates the cost from a node to the goal node

  h>= 0, and h(goal)=0

Admissibility ensures cost optimality - proof

Prove that A* with admissible heuristics cannot return a suboptimal path.

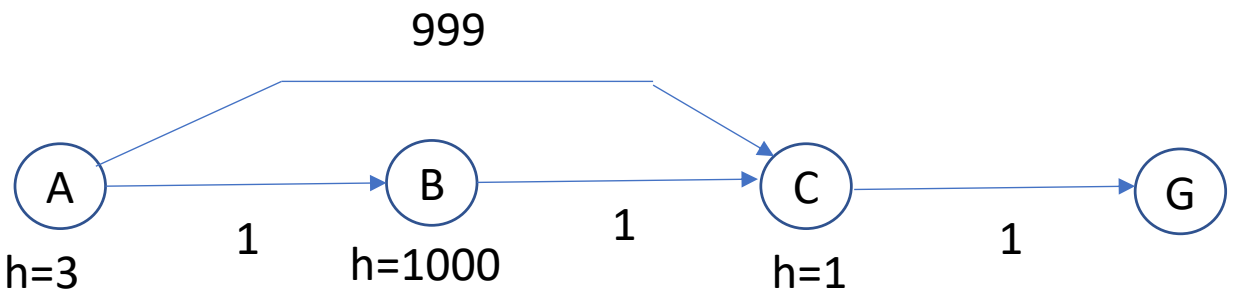
Proof by contradiction:

- Suppose an admissible heuristic returns a suboptimal path.
- Suppose **n** is a node on the optimal path.
- If A* returns a nonoptimal path it means that n was not expanded.
- This means that:

$$
\begin{aligned}
f(n) &> C^\star && \text{otherwise } n \text{ would be expanded} \\
f(n) &= g(n) + h(n) && \text{by definition} \\
f(n) &= g^\star(n) + h(n) && \text{because n is on the optimal path} \\
f(n) &\leq g^\star(n) + h^\star(n) && \text{because of admissibility,} \\
f(n) &\leq C^\star && \text{by definition } \mathrm{C}^\star = \mathrm{g}^\star(\mathrm{n}) + \mathrm{h}^\star(\mathrm{n})
\end{aligned}
$$

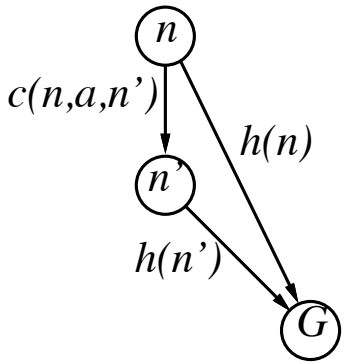
First and last sentence contradict

Example – Admissibility and Cost Optimality



Consistency of heuristic

A heuristic is **consistent** if h(n) $\leq c(n, a, n') + h(n')$



If a heuristic function $h(n)$ is consistent, then the cost function $f(n)$ is monotonic nondecreasing,i.e if and only if for all children n of n ,
f(n') $\geq$ f(n)

Proof:

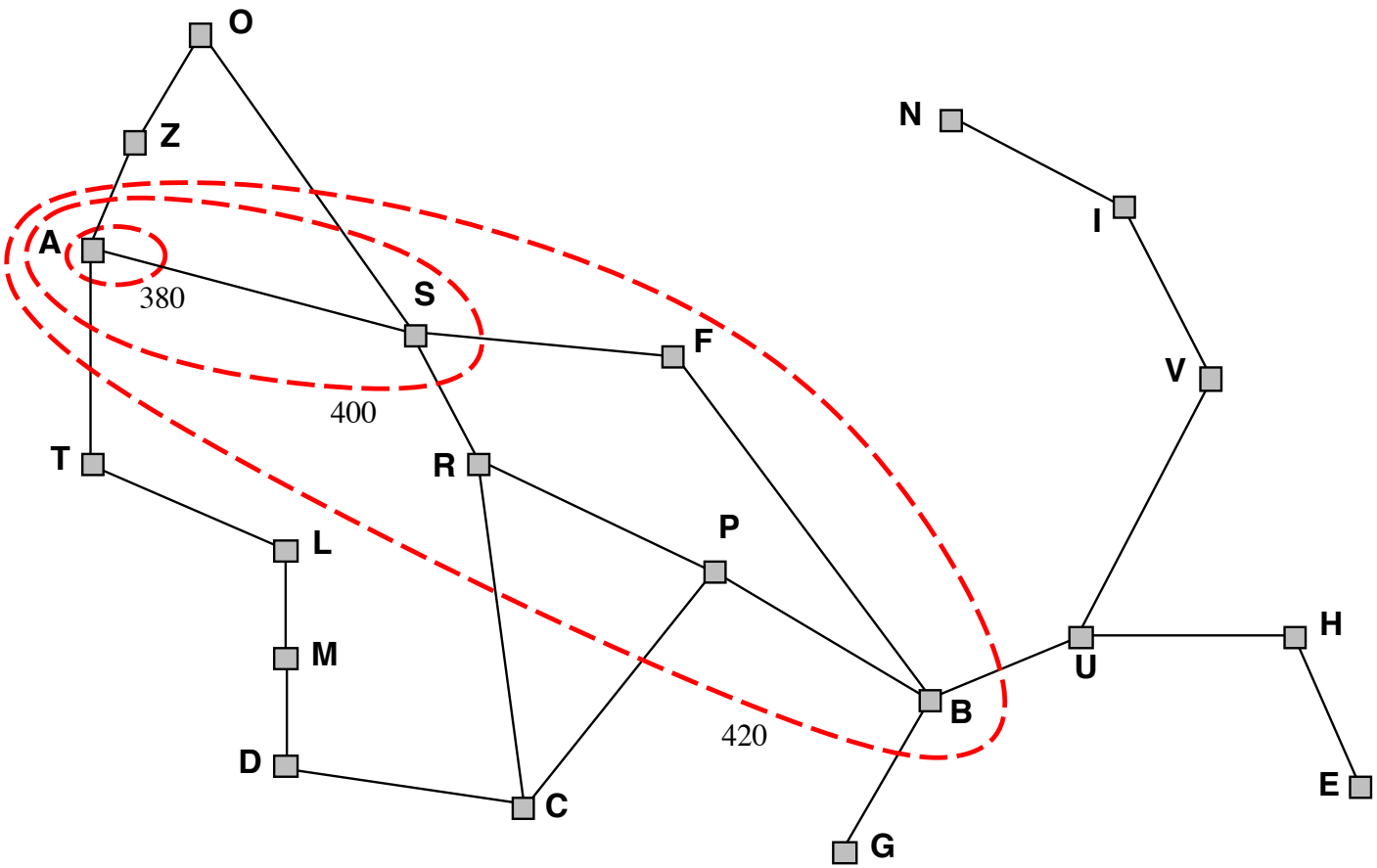
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &\geq f(n) \end{aligned}$$

i.e., $f(n)$ is nondecreasing along any path.

Search contours

Lemma: A* expands nodes in order of increasing f value*

Gradually adds "f-contours" of nodes (like breadth-first adds layers)
Contour i has all nodes with $f \leq f_i$, where $f_i < f_{i+1}$



A * is optimally efficient

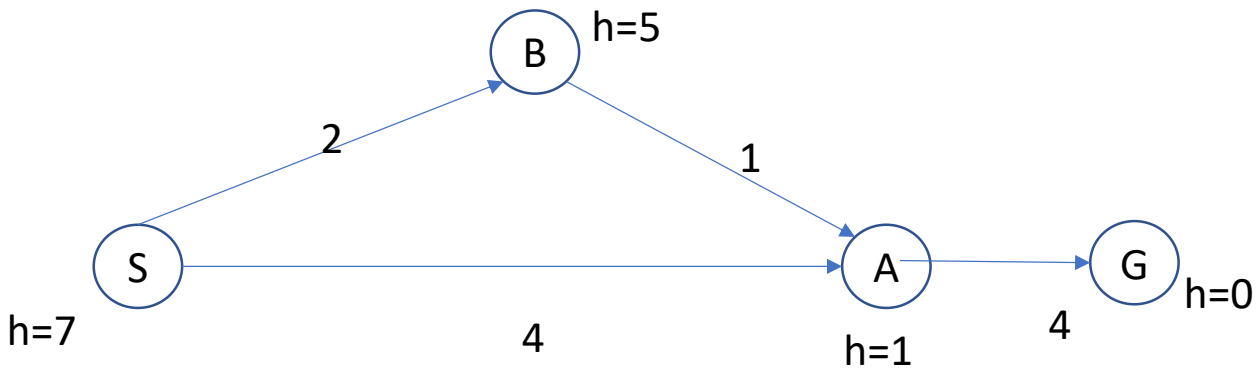
A* search:

- is **optimally efficient** with a consistent heuristic.
- "optimally efficient "means that any other search algorithm with same heuristic values must expand all nodes that are surely expanded by A* "
- "surely" expands all nodes that can be reached from the initial state on a path where every node on the path has $f(n) < C^*$. It opens some of the nodes with $f(n) = C^*$, and no nodes with $f(n) > C^*$.
- doesn't expand f_{i+1} until f_i is finished

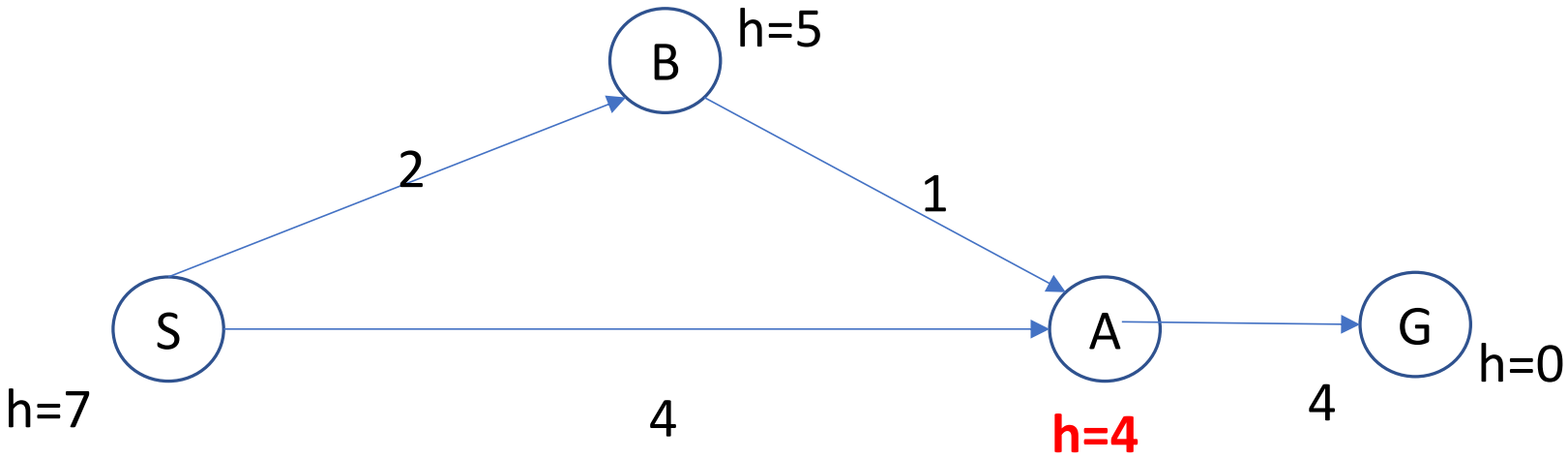
A* is optimally efficient

- consistency is required for optimally efficient search in state spaces with multiple paths to a node.
- with consistent heuristic a state will be reached, the very first time, through the optimal path
- hence, no need for re-expanding nodes
- hence, less number of node expansions

Example - Consistency



Example - Consistency



Memory-bounded search

Main issue of A* is the use of memory.

How to reduce memory use:

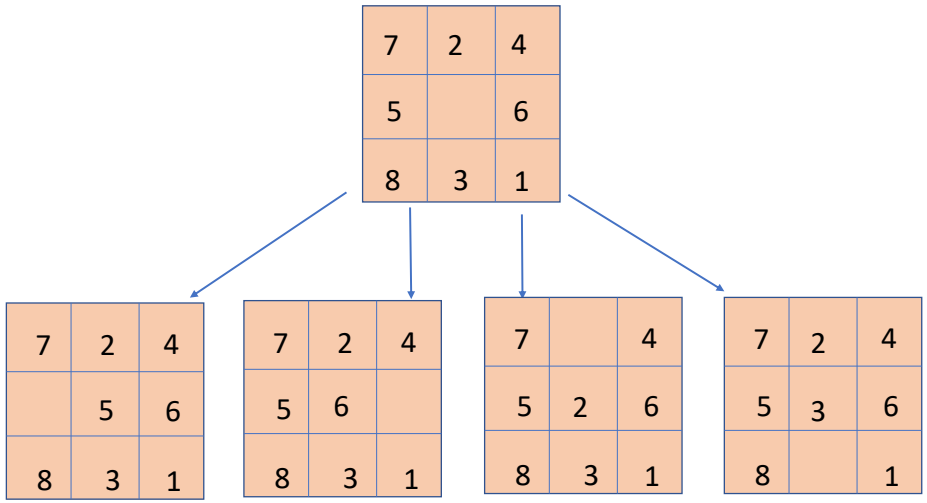
- Reference count - remove a state from *reached* when there are no more ways to reach it
- Beam search - limit size of ıfrontier to k best candidates
- Iterative-deepening A* - gradually increase f-cost cutoff
- Recursive best-first search (RBFS) - like standard best-first serach but with linear space
- Memory-bounded A* - expand until memory is full, then drop the worst
- Bidirectional heuristic search

Heuristic Functions

Admissible heuristics for the 8-puzzle:

Goal state is: Upper left tile is empty, in the rest of the grid: numbers 1-8 are in natural order

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance (i.e., no. of squares from desired location of each tile)

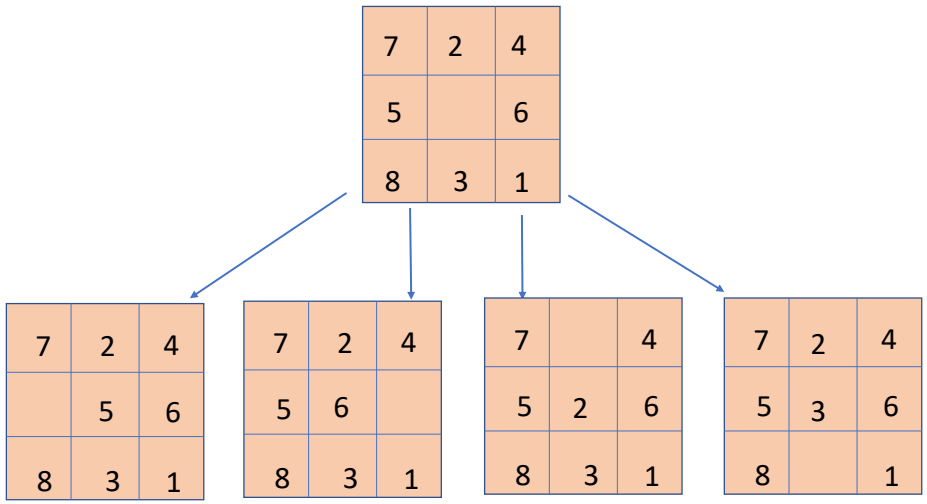


$h_1(S) = ??$
$h_2(S) = ??$

Heuristic Functions

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



$h_1(S) =$?? 8
$h_2(S) =$?? 3+1+2+2+2+3+3+2 = 18
The shortest solution cost is 26 actions long

Effective branching factor

- Effective branching factor ($b^\star$)characterizes the quality of heuristic.
- $b^\star$: Braching factor that a uniform tree of depth d would have to contain $N + 1$ nodes.

  $$\mathrm{N} + 1 = 1 + b^\star + (b^\star)^2 + \ldots + (b^\star)^d$$
- small $b^\star$ (i.e., closer to 1), better heuristic.

Comparison of $b^\star$'s

	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

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,

  $\mathrm{h(n)} = \max(h_a(n), h_b(n))$

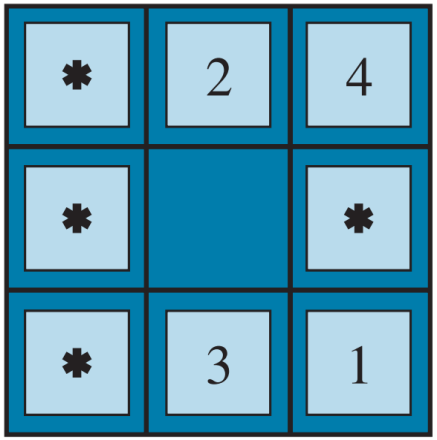
  is also admissible and dominates h_a, h_b

Generating heuristics from relaxed problems

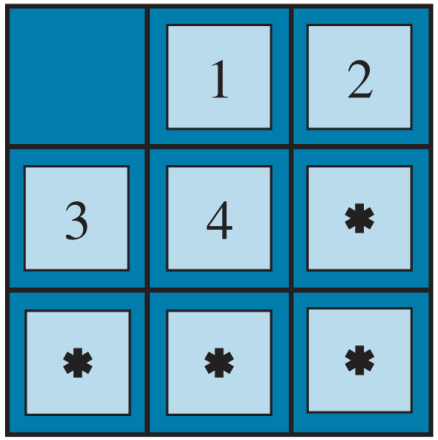
- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem.
- Examples for 8-puzzle:
  - a tile can move to any square
  - a tile can move to any adjacent square
- **Key point**: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Generating heuristics from subproblems

Pattern databases



Start State



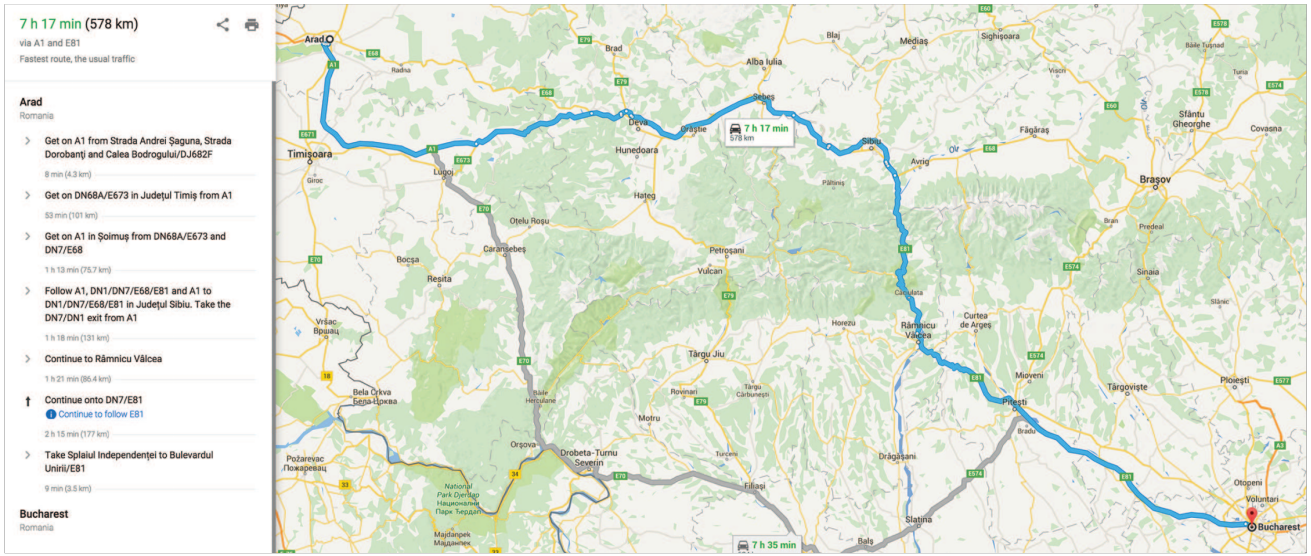
Goal State

$9 \quad 8 \quad 7 \quad 6 \quad 5 = 15{,}120$ patterns in the database

Landmarks heuristic

Generating heuristics with landmarks.

- Precomputation of some optimal path costs
- $h_L(n) = min_{L \in Landmarks} C^*(n, L) + C^*(L, goal)$



Admissible?

Search in complex environments - Chapter 4

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

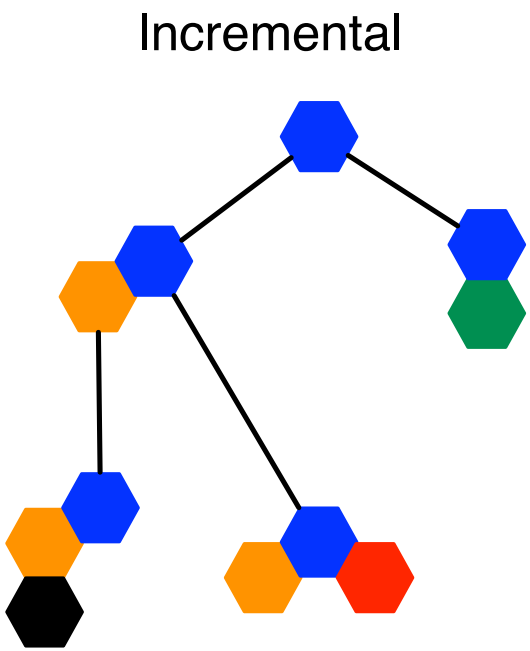
Local Search

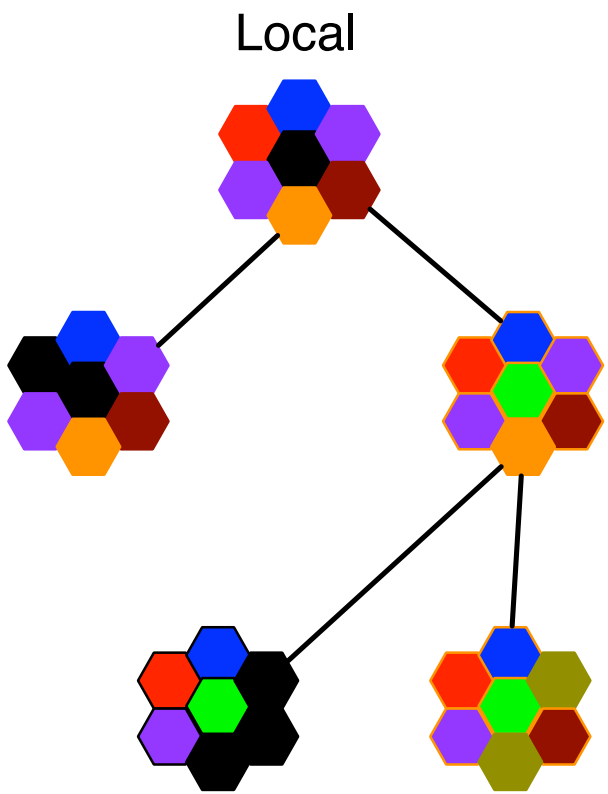
- The uninformed and informed search algorithms are designed to explore search spaces systematically.
  - They keep one or more paths in memory and record which alternatives have been explored at each point along the path
  - The path to that goal constitutes a solution
  - In many problems, however, the path to the goal is irrelevant
- If the path to the goal does not matter $\longrightarrow$ Local search algorithms

Local Search

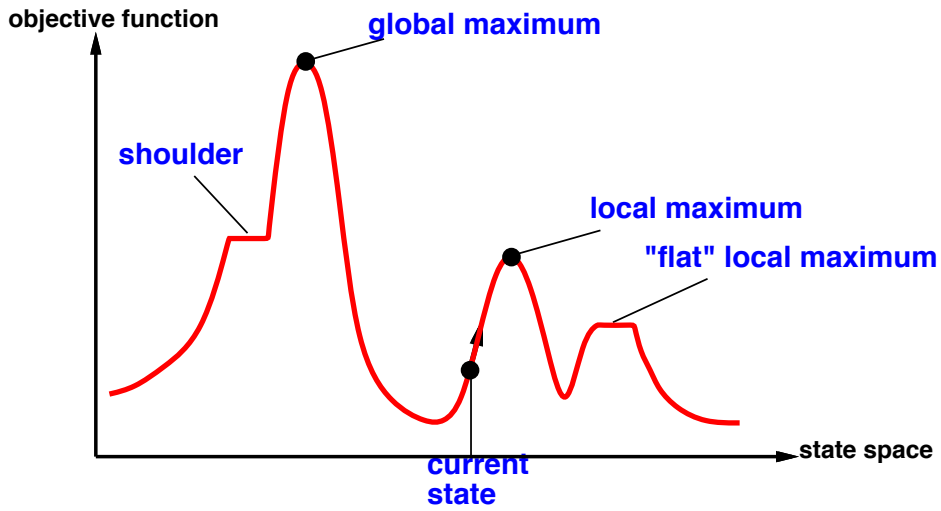
- Often use a single current node and generally move only to neighbours of that node.
- Ease up on completeness and optimality in the interest of improving time and space complexity
- Although local search algorithms are not systematic, they have two key advantages:
  1. use very little memory (usually a constant amount), and
  2. can often find reasonable solutions in large or infinite state spaces.

Incremental versus Local Search





State-space Landscape



Each point in the landscape represent a state in the "world", and has an "elevation".

If the elevation correspond to an objective function, then the aim is to find the highest peak. Then this is **Hill Climbing**

If elevation corresponds to the cost, then the aim is to find the lowest point. This is called **gradient descent.**

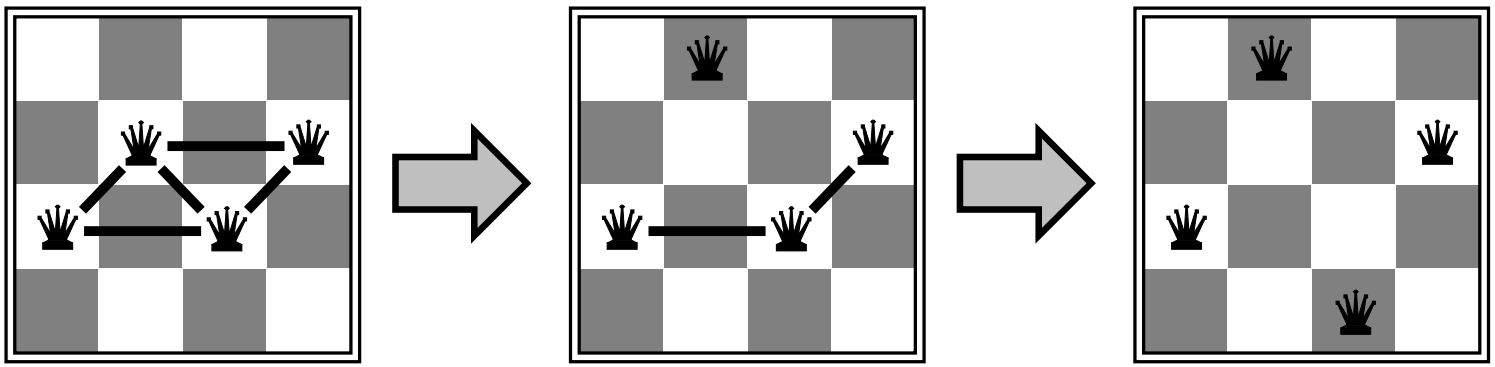
The negative of the cost function can be used as the objective function.

Example:n-queens

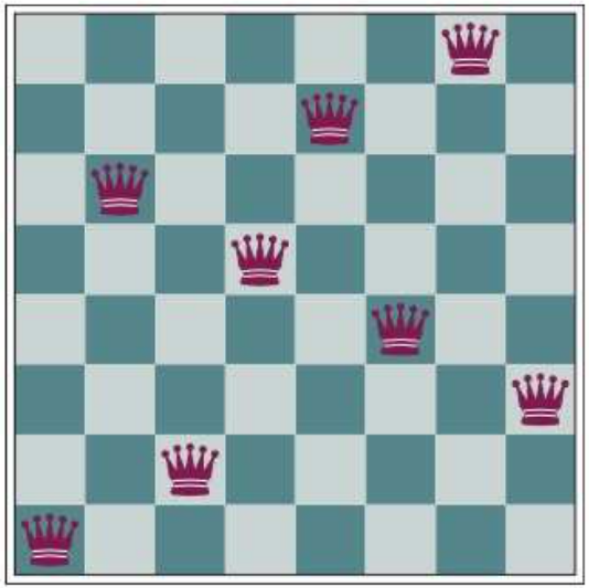
Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Heuristic cost= number of conflicting pairs of queens

Move a queen (on the same column) to reduce number of conflicts.



Example- 8-queens



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

Heuristic cost: nr. of attacking pairs of queens, h=17 for the state on the right.

Hill-climbing

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

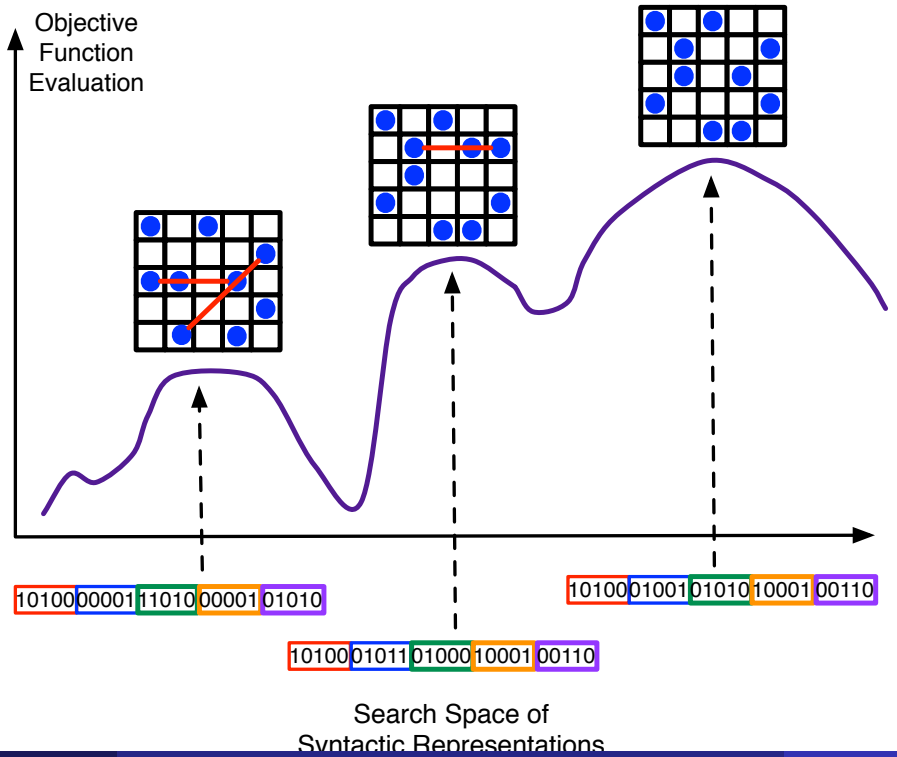
current ← *problem*.INITIAL

while *true* **do**

neighbor ← a highest-valued successor state of *current*

if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*

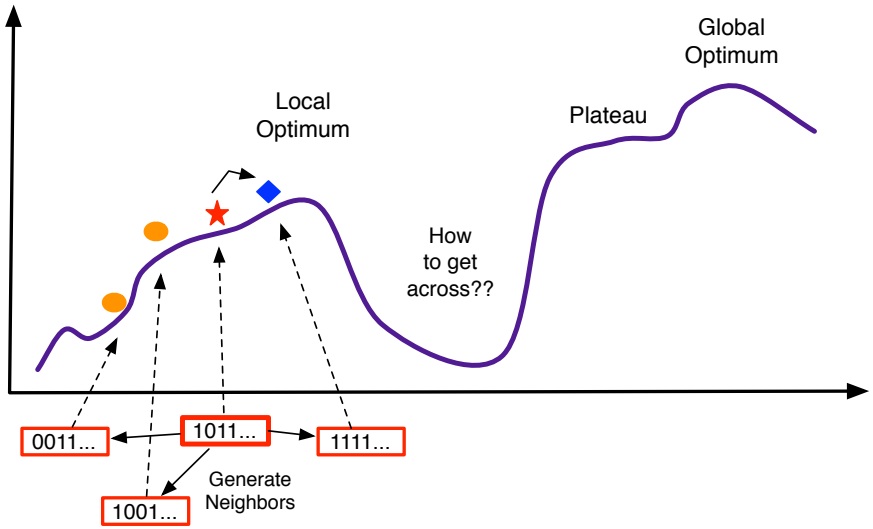
current ← *neighbor*



Hill-climbing

"Like climbing Everest in thick fog with amnesia"

Gets stuck in local maxima, ridges, plateaus



How to not get stuck

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

Simulated annealing

Gradient Descent.

Idea: escape local minima by allowing some "bad" moves but gradually decrease their frequency

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

Local beam search

- Idea: keep k states instead of 1; choose top/good k of all their successors

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

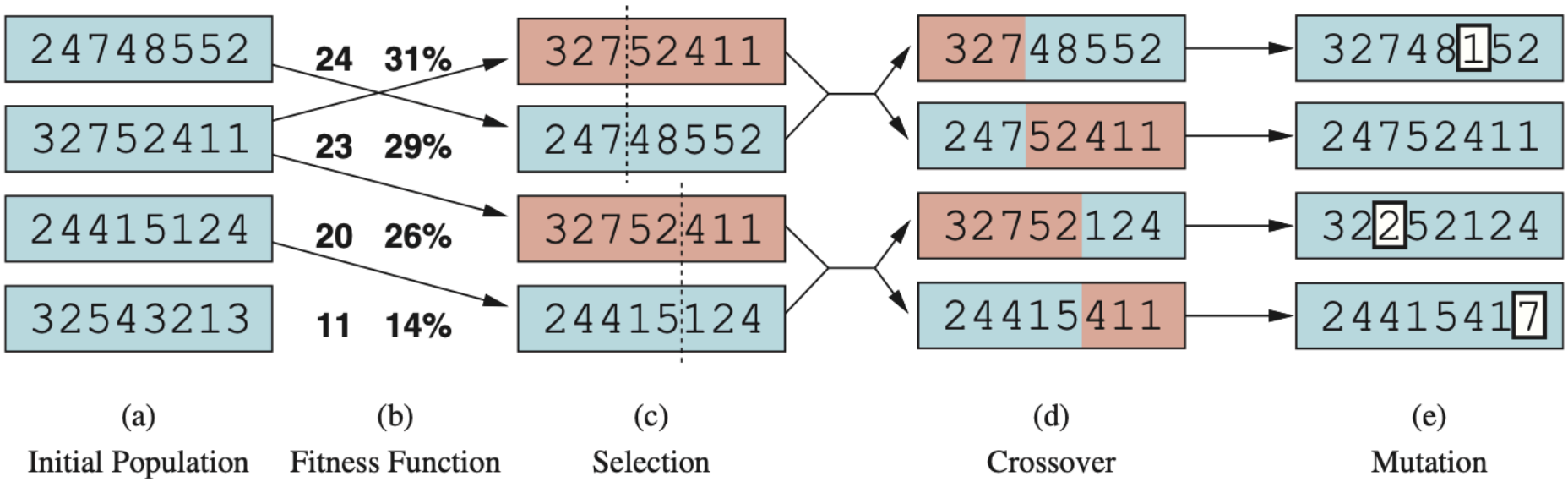
- Problem: quite often, all k states end up on same local hill
- Analogy to natural selection!

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

Example: 8-queens



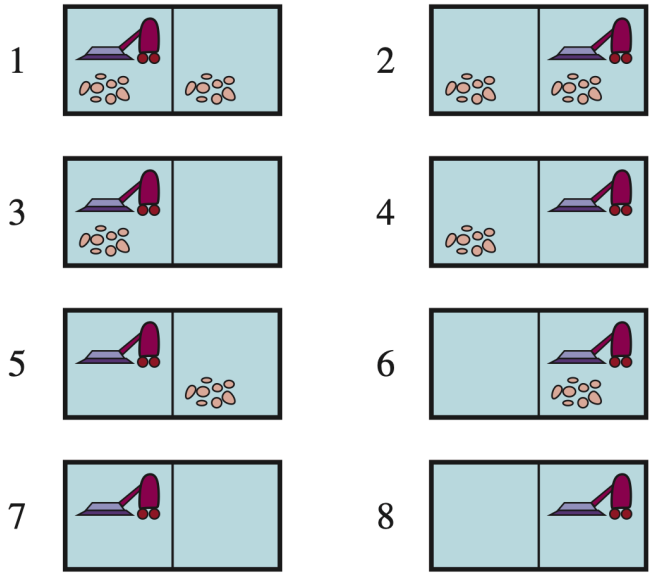
Fitness function (i.e, objective fn): number of non-attacking pairs
Selection: Probability of being regenerated in the next generation

Searching with nondeterministic and not fully obervable environments

- So far, we have assumed that the actions are deterministic, the environment is fully observable and known.
- In the real-world, things do not always go as expected.
- To account for different possible outcomes, we need to come up with a contingency plan instead of a single path of actions.

Searching with nondeterministic actions

Example - the erratic vacuum world



Non-deterministic suck action:

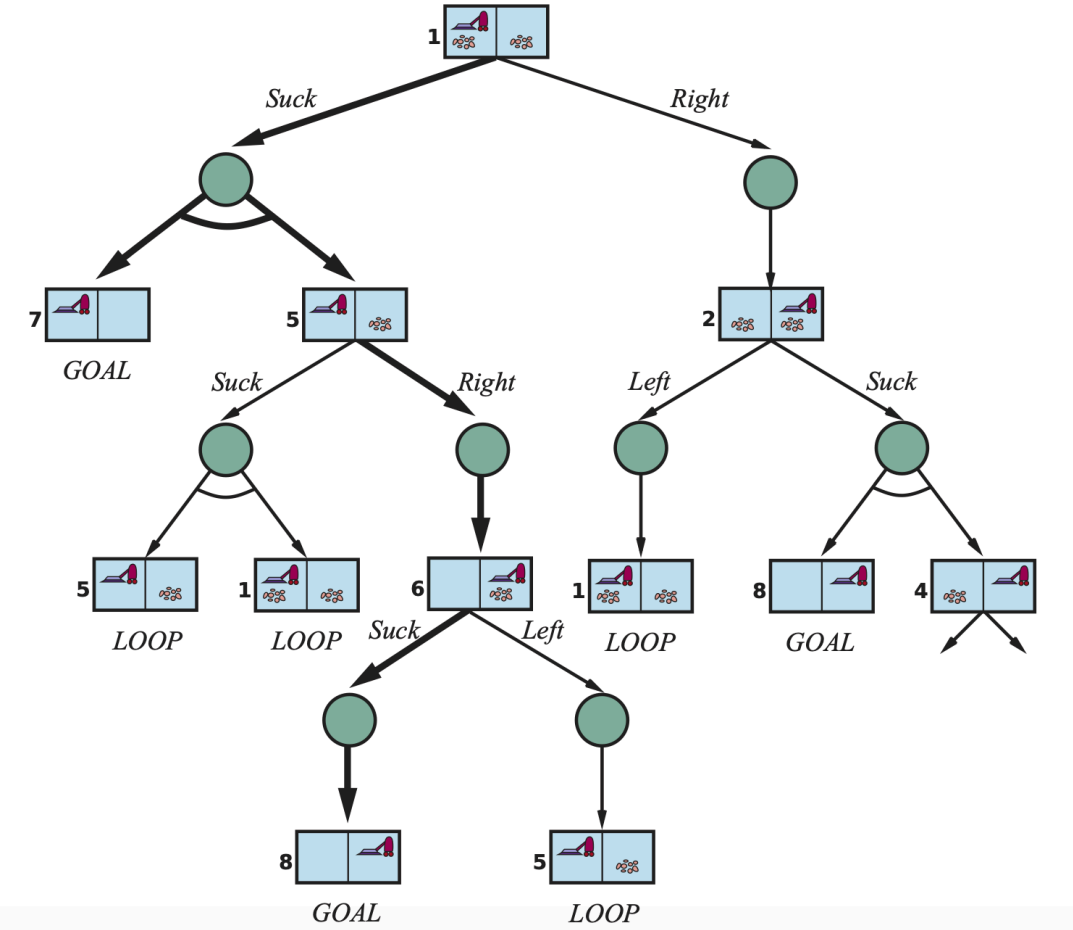
- When applied to a dirty square, it cleans the square and sometimes cleans an adjacent square too.
  e.g. Results(1, Suck) = {5,7}
- When applied to a clean square, it may deposit dirt on the square. e.g. Results(7, Suck) = {7,3}

Solution for initial state 1 is a contingency plan:
[*Suck*, *if State* = 5 *then* [*Right*, *Suck*] *else* []]

AND-OR search trees

OR nodes - actions.

AND nodes - outcomes



Solution for erratic vacuum cleaner, contingency plan:
[*Suck*, *if* *State* = 5 *then* [*Right*, *Suck*] *else* []]

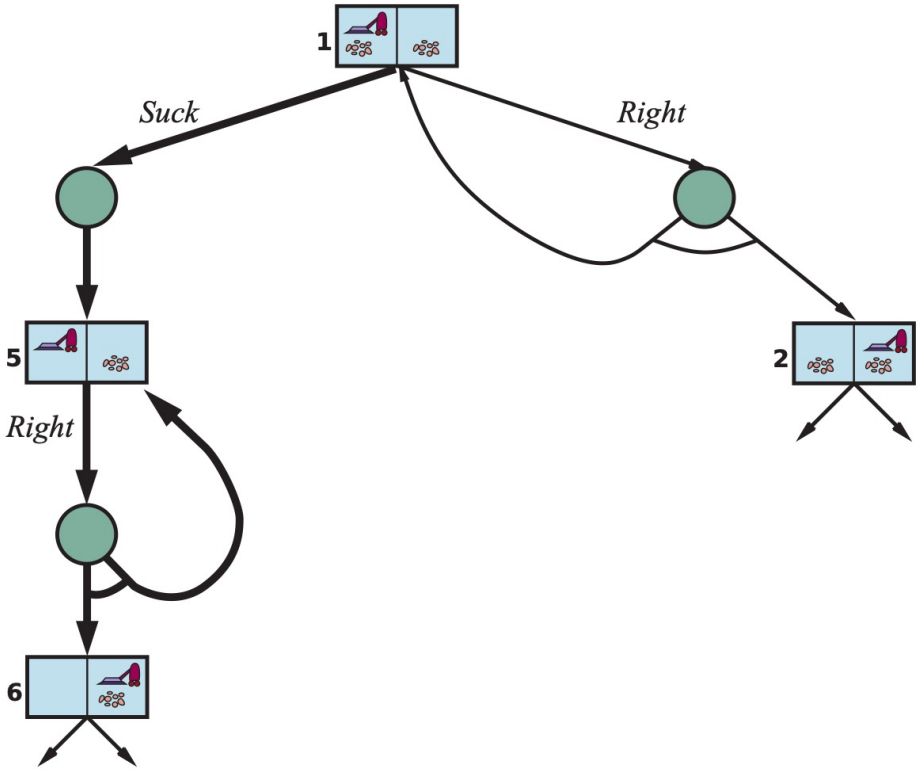
Recusive, depth-first and-or graph search

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**
    $plan \leftarrow$ AND-SEARCH(*problem*, RESULTS(*state*, *action*), [*state*] + *path*])
    if $plan \neq failure$ **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**
    $plan_i \leftarrow$ 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** $plan_n$]

Loops and Cyclic plans - Slippery VC

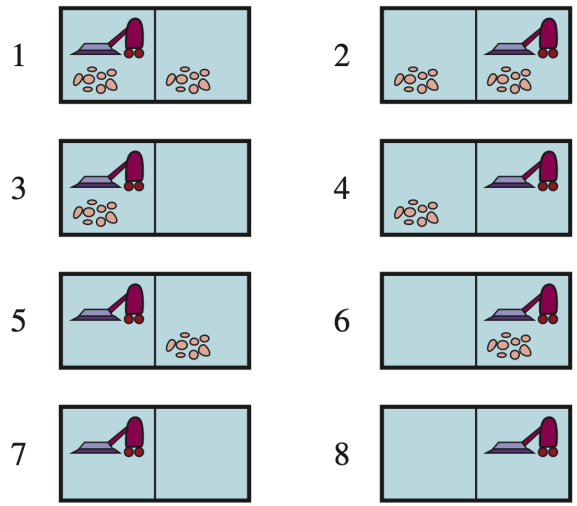


- **Example**: Slippery vacuum world
- Movement actions sometimes fail. e.g. Results(1, Right) = {1,2}
- Cyclic solution: [*Suck*, *while State* = 5 *do Right*, *Suck*]

Searching in Partially Observable Environments

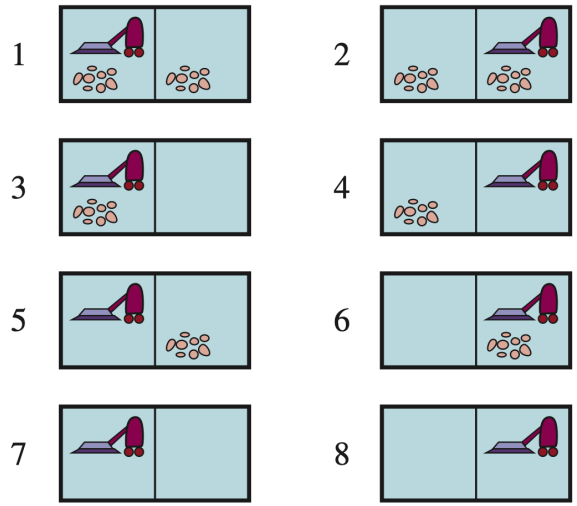
- So far, we have assumed that the agent knows exactly the state of its environment.
- In reality, an agent receives partial, possibly noised, observations.
- Therefore, the state can only be estimated - "belief state space".
- Solution for entirely sensorless problems: a sequence of actions
- Solution for a "partially-observing sensor": a contingency plan.

Sensorless (deterministic) vacuum world



- Initial state: A belief state comprising all physical states, {1,2,3,4,5,6,7,8}
- Solution: a sequence of actions - since no perception

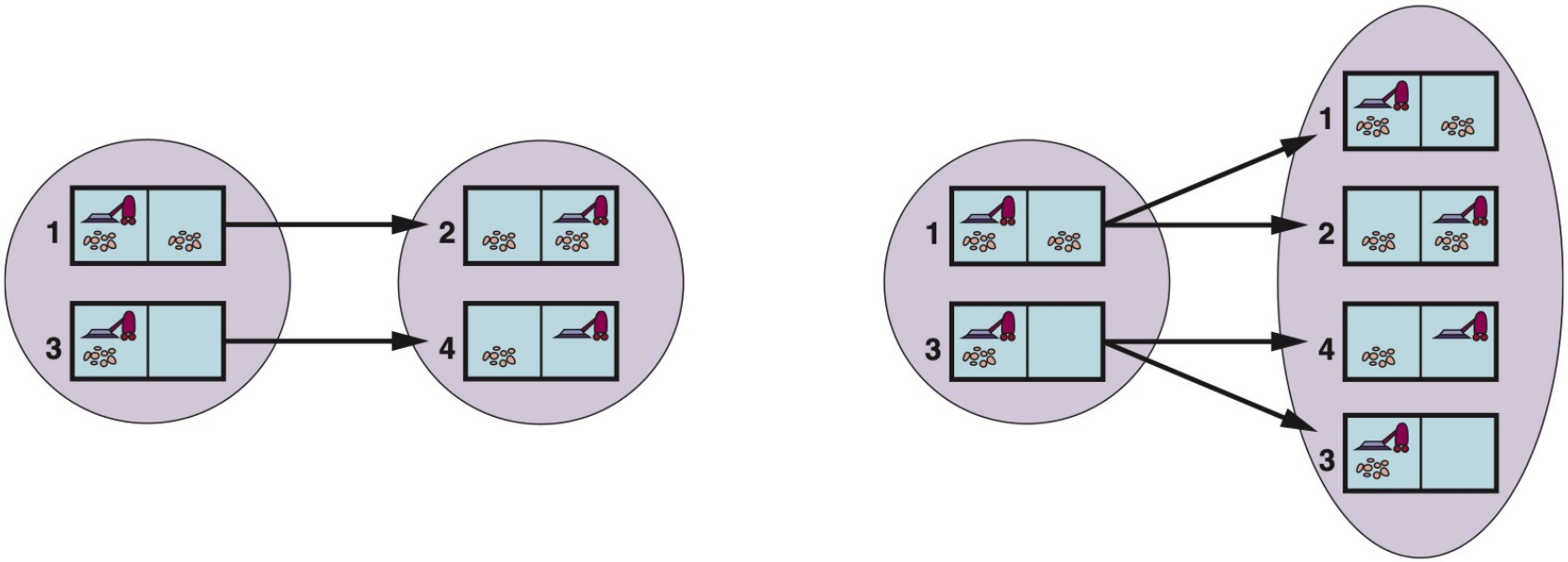
Sensorless deterministic vacuum world



Transition model:

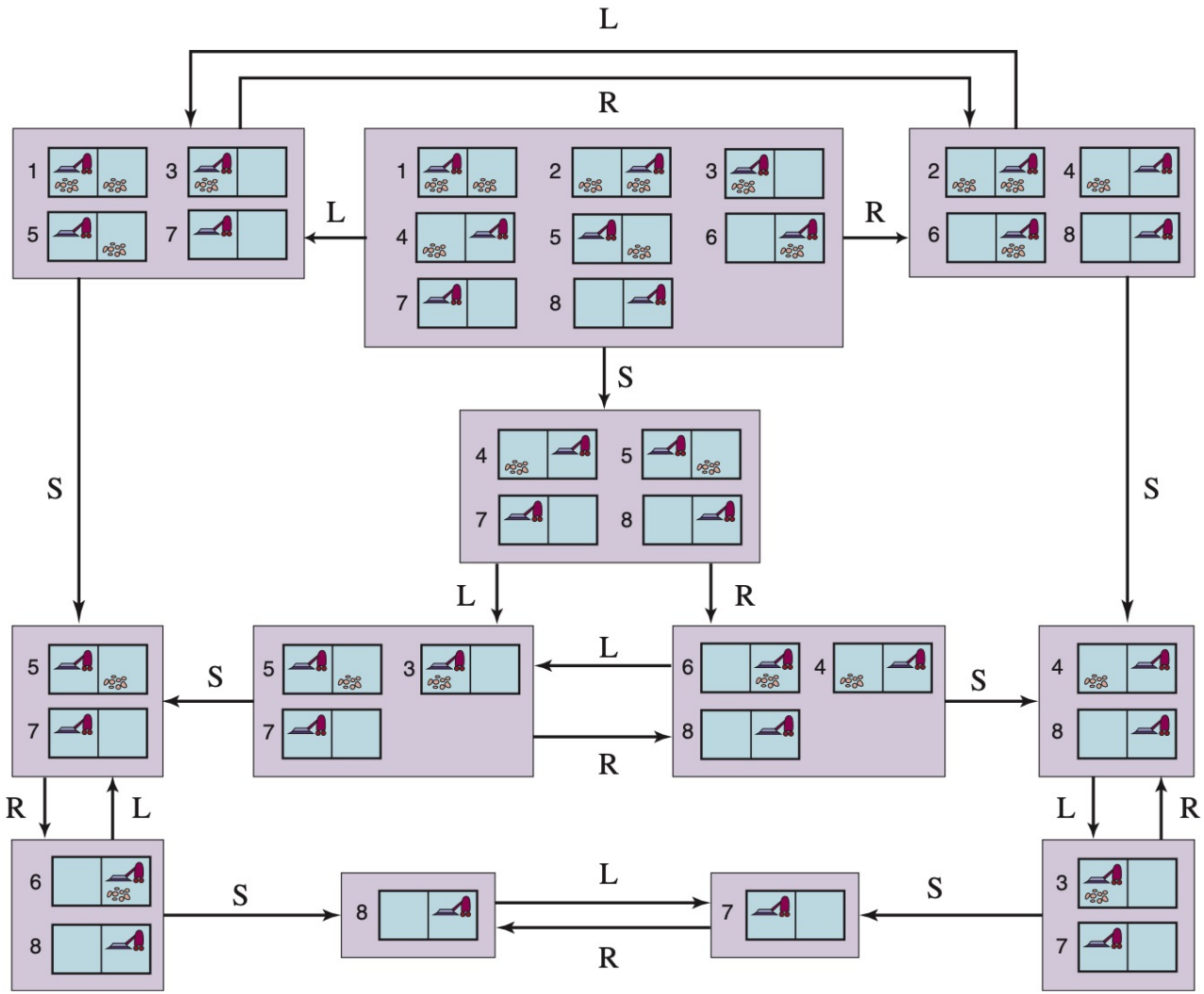
1. Result({1,2,3,4,5,6,7,8}, Right) = {2,4,6,8}
2. Result({2,4,6,8}, Suck) = {4,8}
3. Result({4,8}, Left) = {1,7}
4. Result({1,7}, Suck) = {7}

Predicting the next state with sensorless agents



Predicting the next belief state when the action *Right* is taken in deterministic (left) and non-deterministic (right)situations

Belief state space for sensorless, deterministic VC



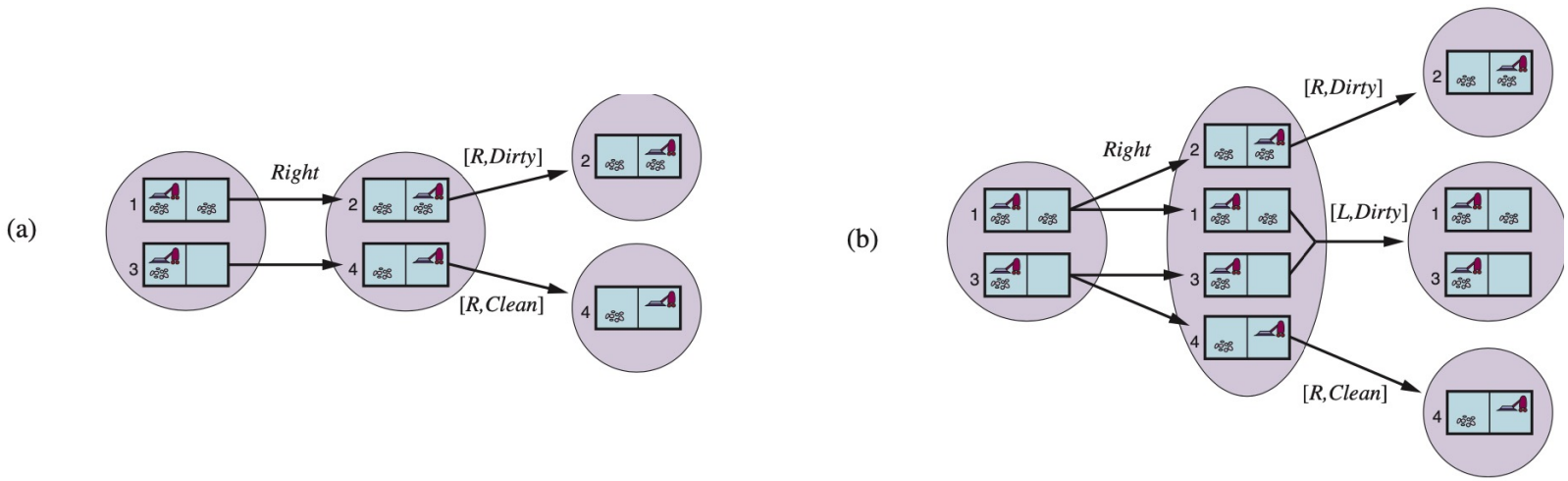
Ordinary search algorithms can be used to search this belief-state space.

Search in partially observable environments

Partially observable environment: Vacuum agent sees only part of the environment, the square it is in and whether dirt/not.

Transition model:

- Prediction - similar to RESULT fn, output is a "belief state"
- Possible percepts - set of percept that can be observed in the predicted belief state
- Update - for each possible percept, computes the belief state that would result from the percept

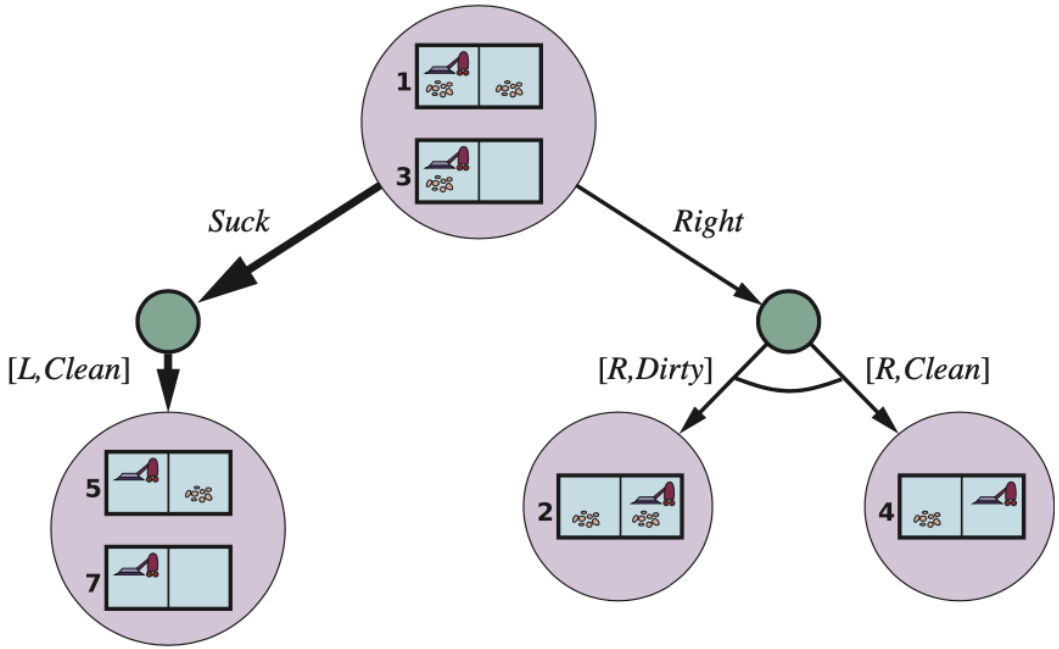


(a) In deterministic world, (b) Non-deterministic world.

Search in partially observable environments

Using AND-OR search.

Suppose the Initial percept is $[L, dirty]$



Notice that a solution is a conditional plan - because there is perception in local-sensing agents.

Summary

- heuristic
- A* search algorithm
- local search algorithms
- search in partially observable and nondeterministic environments

- **Next week: Chapter 5 - Constraint Satisfaction Problems**