Fundamentals of Artificial Intelligence
Chapter 04: **Beyond Classical Search**

Roberto Sebastiani

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it
http://disi.unitn.it/rseba/DIDATTICA/fai_2020/

Teaching assistant: Mauro Dragoni – dragoni@fbk.eu
http://www.maurodragoni.com/teaching/fai/

M.S. Course "Artificial Intelligence Systems", academic year 2020-2021

Last update: Tuesday 8th December, 2020, 13:06

Outline

Outline

1. Local Search and Optimization
   - Hill-Climbing
   - Simulated Annealing
   - Local Beam Search & Genetic Algorithms
2. Local Search in Continuous Spaces [hints]
3. Search with Nondeterministic Actions
4. Search with Partial Observations
   - Search with No Observations
   - Search with (Partial) Observations
5. Online Search

General Ideas

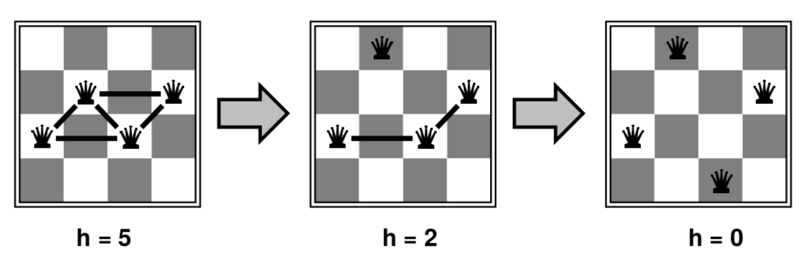
- Search techniques: systematic exploration of search space
  - solution to problem: the path to the goal state
  - ex: 8-puzzle
- With many problems, the path to goal is irrelevant
  - solution to problem: only the goal state itself
  - ex: N-queens
  - many important applications: integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, portfolio management...
  - goals expressed as conditions, not as explicit list of goal states
- The state space is a set of “complete” configurations
  - find goal configuration satisfying constraints/rules (ex: N-queens)
  - find optimal configurations (ex: Travelling Salesperson Problem, TSP)
- If so, we can use iterative-improvement algorithms (in particular local search algorithms):
  - keep a single “current” state, try to improve it

Local Search

- Idea: use single current state and move to "neighbouring" states
  - operate using a single current node
  - the paths followed by the search are not retained
- Two key advantages:
  - use very little memory (usually constant)
  - can often find reasonable solutions in large or infinite (continuous) state spaces, for which systematic algorithms are unsuitable
- Also useful for pure optimization problems
  - find the best state according to an objective function
  - often do not fit the "standard" search model of previous chapter
  - ex: Darwinian survival of the fittest: metaphor for optimization, but no "goal test" and no "path cost"
- A complete local search algorithm: guaranteed to always find a solution (if exists)
- A optimal local search algorithm: guaranteed to always find a maximum/minimum solution
  - maximization and minimization dual (switch sign)

Local Search Example: N-Queens

- One queen per column (incremental representation)
- Cost (h): # of queen pairs on the same row, column, or diagonal
- Goal: h=0
- Step: move a queen vertically to reduce number of conflicts



(© S. Russell & P. Norwig, AIMA)

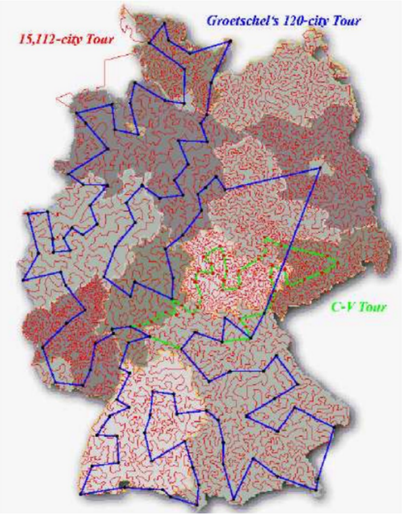
Almost always solves N-queens problems almost instantaneously for very large N (e.g., N=1million)

Optimization Local Search Example: TSP

Travelling Salesperson Problem (TSP)

Given an undirected graph, with n nodes and each arc associated with a positive value, find the Hamiltonian tour with the minimum total cost.

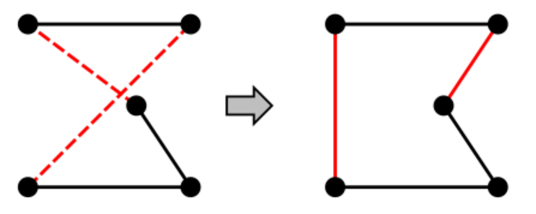
Very hard for classic search!



(Courtesy of Michela Milano, UniBO)

Optimization Local Search Example: TSP

- State represented as a permutation of numbers $(1, 2, ..., n)$
- Cost (h): total cycle length
- Start with any complete tour
- Step: (2-swap) perform pairwise exchange

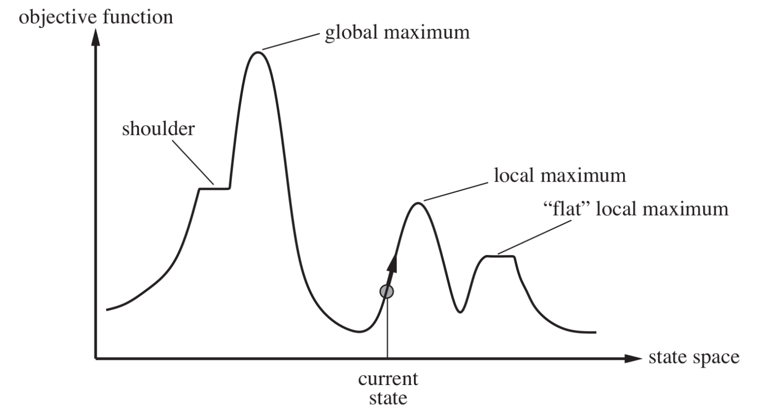


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Local Search: State-Space Landscape

State-space landscape (Maximization)

- Local search algorithms explore state-space landscape
  - state space n-dimensional (and typically discrete)
  - move to "nearby" states (neighbours)
- NP-Hard problems may have exponentially-many local optima



(© S. Russell & P. Norwig, AIMA)

Outline

Hill-Climbing Search (aka Greedy Local Search)

Hill-Climbing

- Very-basic local search algorithm
- Idea: a move is only performed if the solution it produces is better than the current solution
  - (steepest-ascent version): selects the neighbour with best score improvement (select randomly among best neighbours if ≥ 1)
  - does not look ahead of immediate neighbors of the current state
  - stops as soon as it finds a (possibly local) minimum
- Several variants (Stochastic H.-C., Random-Restart H.-C., ...
- Often used as part of more complex local-search algorithms

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

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
     neighbor ← a highest-valued successor of current
     if neighbor.VALUE ≤ current.VALUE then return current.STATE
     current ← neighbor
```

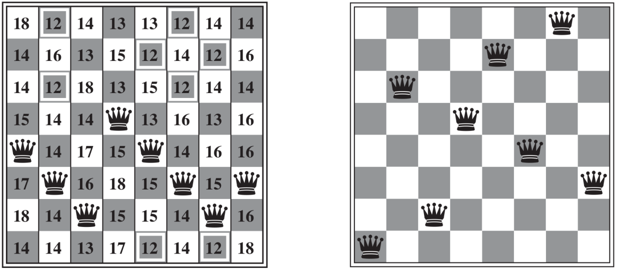
Hill-Climbing Search: Example

8-queen puzzle (minimization)

- Neighbour states: generated by moving one queen vertically
  - Cost (h): # of queen pairs on the same row, column, or diagonal
  - Goal: h=0
- Two scenarios ((*a*) $\Longrightarrow$ (*b*) in 5 steps) :

(*a*) 8-queens state with heuristic cost estimate h = 17 (12d, 5h)

(*b*) local minimum: h=1, but all neighbours have higher costs



(a) (b)

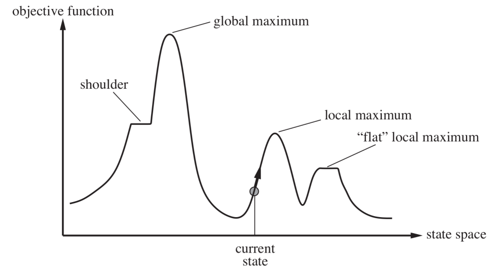
(© S. Russell & P. Norwig, AIMA)

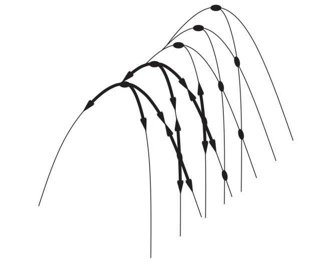
Hill-Climbing Search: Drawbacks

- Incomplete: gets stuck in local optima, flat local optima & shoulders (aka plateaux), ridges (sequences of local optima)
  - Ex: with 8-queens, gets stuck 86% of the time, fast when succeed

  note: converges very fast till (local) minima or plateaux
- Possible idea: allow 0-progress moves (aka sideways moves)
  - pros: may allow getting out of shoulders
  - cons: may cause infinite loops with flat local optima

  $\Longrightarrow$ set a limit to consecutive sideways moves (e.g. 100)
  - Ex: with 8-queens, pass from 14% to 94% success, slower





Hill-climbing: Variations

- Stochastic hill-climbing
  - random selection among the uphill moves
  - selection probability can vary with the steepness of uphill move
  - sometimes slower, but often finds better solutions
- First-choice hill-climbing
  - cfr. stochastic h.c., generates successors randomly until a better one is found
  - good when there are large amounts of successors
- Random-restart hill-climbing
  - conducts a series of hill-climbing searches from randomly generated initial states
  - Tries to avoid getting stuck in local maxima

Outline

Simulated Annealing

- Inspired to statistical-mechanics analysis of metallurgical annealing (Boltzmann's state distributions)
- Idea: Escape local maxima by allowing "bad" moves...
  - "bad move": move toward states with worse value
  - typically pick a move taken at random ("random walk")
- ... but gradually decrease their size and frequency.
  - sideways moves progressively less likely
- Analogy: get a ball into the deepest crevice in a bumpy surface
  - initially shaking hard ("high temperature")
  - progressively shaking less hard ("decrease the temperature")

Widely used in VSLI layout problems, factory scheduling, and other large-scale optimization tasks

Simulated Annealing [cont.]

Simulated Annealing (maximization)

- "temperature" T slowly decreases with steps ("schedule")
- The probability of picking a "bad move":
  - decreases exponentially with the "badness" of the move $|\Delta E|$
  - decreases as the "temperature" T goes down
- If schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
          schedule, a mapping from time to "temperature"

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

Outline

Local Beam Search

Local Beam Search

- Idea: keep track of k states instead of one
- Initially: k random states
- Step:
  1. determine all successors of k states
  2. if any of successors is goal $\Longrightarrow$ finished
  3. else select k best from successors
- Different from k searches run in parallel:
  - searches that find good states recruit other searches to join them $\Longrightarrow$ information is shared among k search threads
- Lack of diversity: quite often, all k states end up same local hill

$\Longrightarrow$ Stochastic Local Beam: choose k successors randomly, with probability proportionally to state success.

Resembles natural selection with asexual reproduction:
the successors (offspring) of a state (organism) populate the next generation according to its value (fitness), with a random component.

Genetic Algorithms

- Variant of local beam search: successor states generated by combining two parent states (rather than one single state)
- States represented as strings over a finite alphabet (e.g. $\{0, 1\}$)
- Initially: pick k random states
- Step:
  1. parent states are rated according to a fitness function
  2. k parent pairs are selected at random for reproduction, with probability increasing with their fitness
     - gender and monogamy not considered
  3. for each parent pair
     1. a crossover point is chosen randomly
     2. a new state is created by crossing over the parent strings
     3. the offspring state is subject to (low-probability) random mutation
- Ends when some state is fit enough (or timeout)
- Many algorithm variants available

Resembles natural selection, with sexual reproduction

Genetic Algorithms

function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual
  inputs: *population*, a set of individuals
    FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 **to** SIZE(*population*) **do**
      x ← RANDOM-SELECTION(*population*, FITNESS-FN)
      y ← RANDOM-SELECTION(*population*, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
    population ← *new_population*
  until some individual is fit enough, or enough time has elapsed
  return the best individual in *population*, according to FITNESS-FN

---

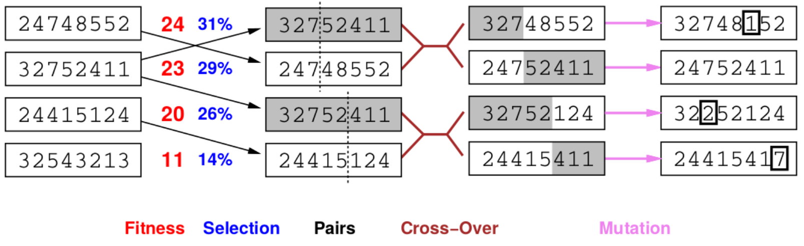
function REPRODUCE(x, y) **returns** an individual
  inputs: x, y, parent individuals

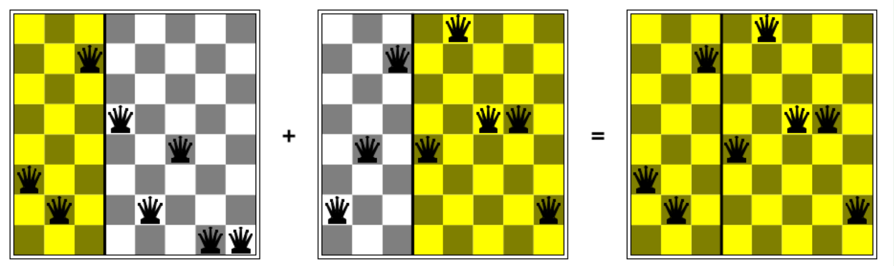
  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING($x, 1, c$), SUBSTRING($y, c + 1, n$))

Genetic Algorithms: Example

Example: 8-Queens

state[*i*]: (upward) position of the queen in *i*th column





(© S. Russell & P. Norwig, AIMA)

Genetic Algorithms: Intuitions, Pros & Cons

Intuitions

- Selection drives the population toward high fitness
- Crossover combines good parts from good solutions (but it might achieve the opposite effect)
- Mutation introduces diversity

Pros & Cons

- Pros:
  - extremely simple
  - general purpose
  - tractable theoretical models
- Cons:
  - not completely understood
  - good coding is crucial (e.g., Gray codes for numbers)
  - too simple genetic operators

Widespread impact on optimization problems, i.e. circuit layout and job-shop scheduling

Outline

Local Search in Continuous Spaces [Hints]

Continuous environments

- Successor function produces infinitely many states
  $\Longrightarrow$ previous techniques do not apply
- Discretize the neighborhood of each state
  - turn continuous space into discrete space
  - e.g., empirical gradient considers $\pm\delta$ change in each coordinate
- Gradient methods compute gradients

$$\nabla f \stackrel{\text{def}}{=} \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_k}\right]$$

  to increase/reduce f, e.g. by $x := x + \nabla f(x)$
- The Newton/Raphson Method iterates $x := x - H_f^{-1}(X)\nabla f(x)$
  where $H_f[i,j] \stackrel{\text{def}}{=} \frac{\partial^2 f}{\partial x_i \partial x_j}$ (Hessian matrix)
- all techniques from optimization theory

Outline

Generalities

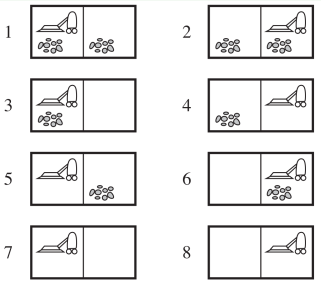
- Assumptions so far (see ch. 2 and 3):
  - the environment is deterministic
  - the environment is fully observable
  - the agent knows the effects of each action

$\Longrightarrow$ The agent does not need perception:
  - can calculate which state results from any sequence of actions
  - always knows which state it is in
- If one of the above does not hold, then percepts are useful
  - the future percepts cannot be determined in advance
  - the agent's future actions will depend on future percepts
- Solution: not a sequence but a contingency plan (aka conditional plan, strategy)
  - specifies the actions depending on what percepts are received
- We analyze first the case of nondeterministic environments

Example: The Erratic Vacuum Cleaner

Erratic Vacuum-Cleaner Example

- actions: Left, Right, Suck
- goal: A and B cleaned (states 7, 8)
- if environment is observable, deterministic, and completely known $\Longrightarrow$ solvable by search algos
- ex: if initially in 1, then [suck,right,suck] leads to 8: [1,5,6,8]



(© S. Russell & P. Norwig, AIMA)

- Nondeterministic version (erratic vacuum cleaner):
  - if dirty square: cleans the square, sometimes cleans also the other square. Ex: $1 \stackrel{suck}{\Longrightarrow} \{5, 7\}$
  - if clean square: sometimes deposits dirt on the carpet Ex: $5 \stackrel{suck}{\Longrightarrow} \{1, 5\}$

Searching with Nondeterministic Actions

Generalized notion of transition model

- RESULTS(S,A) returns a set of possible outcomes states
  - Ex: RESULTS(1,SUCK)=$\{5,7\}$, RESULTS(5,SUCK)=$\{1,5\}$, ...
- A solution is a contingency plan (aka conditional plan, strategy)
  - contains nested conditions on future percepts (if-then-else, case-switch, ...)
  - Ex: from state 1 we can act the following contingency plan: [SUCK, IF STATE = 5 THEN [RIGHT, SUCK] ELSE [ ]]
- Can cause loops (see later)

Searching with Nondeterministic Actions [cont.]

And-Or Search Trees

- In a deterministic environment, branching on agent's choices

  $\Longrightarrow$ OR nodes, hence OR search trees
  - OR nodes correspond to states
- In a nondeterministic environment, branching also on environment's choice of outcome for each action
  - the agent has to handle all such outcomes

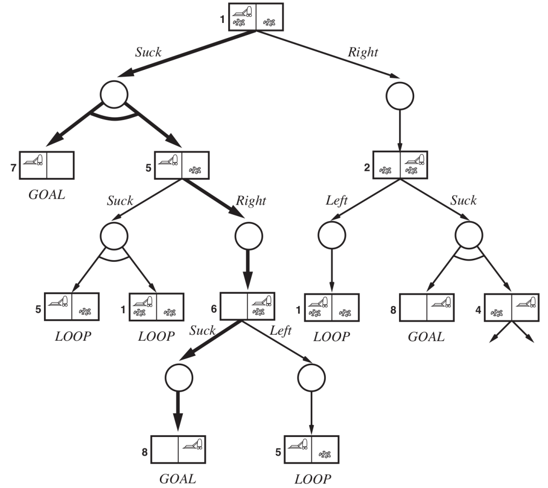
  $\Longrightarrow$ AND nodes, hence AND-OR search trees
  - AND nodes correspond to actions
  - leaf nodes are goal, dead-end or loop OR nodes
- A solution for an AND-OR search problem is a subtree s.t.:
  - has a goal node at every leaf
  - specifies one action at each of its OR nodes
  - includes all outcome branches at each of its AND nodes

OR tree: AND-OR tree with 1 outcome each AND node (determinism)

And-Or Search Trees: Example

(Part of) And-Or Search Tree for Erratic Vacuum Cleaner Example.

Solution for [SUCK, IF STATE = 5 THEN [RIGHT, SUCK] ELSE [ ]]



(© S. Russell & P. Norwig, AIMA)

AND-OR Search

Recursive Depth-First (Tree-based) AND-OR Search

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

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

```
function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
   for each s_i in states do
       plan_i ← OR-SEARCH(s_i, problem, 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]
```


Note: nested if-then-else can be rewritten as case-switch

AND-OR Search [cont.]

Recursive Depth-First (Tree-based) AND-OR Search

- Cycles: if the current state already occurs in the path $\Longrightarrow$ failure
  - cycle detection like with ordinary DFS
  - does not mean "no solution"
  - means "if there is a non-cyclic solution, it must be reachable from the earlier incarnation of the current state"

$\Longrightarrow$ Complete (if state space finite): every path must reach a goal, a dead-end or loop state

- Can be augmented with "explored" data structure for avoiding redundant branches (graph-based search)
- Can also be explored by breadth-first or best-first method
  - e.g. A^* variant for AND-OR search available (see AIMA book)

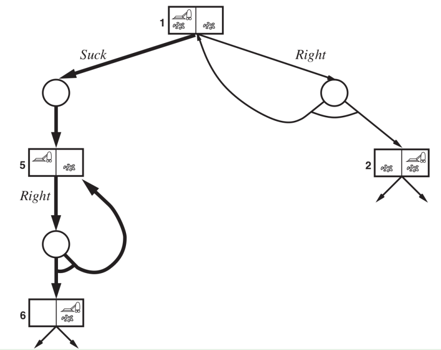
AND-OR Search: Cyclic Solutions

- Some problems have no acyclic solutions
- A cyclic plan may be considered a cyclic solution provided that:
  - every leaf is a goal state(loop states not considered leaves), and
  - a leaf is reachable from every point in the plan
- Can be expressed by means of introducing
  - labels, and backward goto's to labels
  - loop syntax (e.g., while-do)
- Executing a cyclic solution eventually reaches a goal, provided that each outcome of a nondeterministic action eventually occurs
- Is this assumption reasonble?
- Yes, provided we distinguish: $\langle$*nondeterministic*, *observable*$\rangle$ $\neq \langle$*deterministic*, *partially-observable*$\rangle$
- Ex: device may not always work $\neq$ device is broken (but we don't know it)

Cyclic Solution: Example

Example: Slippery Vacuum Cleaner

- Movement actions may fail: e.g., $\mathit{Results}(1, \mathit{Right}) = \{1, 2\}$
- A cyclic solution
- Use labels: [Suck, L1 : Right, if State = 5 then L1 else Suck]
- Use cycles: [Suck, While State = 5 do Right, Suck]



Outline

Generalities

Partial Observability & Belief States

- Partial observability: percepts do not capture the whole state
  - partial state corresponds to a set of possible physical states
- If the agent is in one of several possible physical states, then an action may lead to one of several possible outcomes, even if the environment is deterministic
- Belief state: the agent's current belief about the possible physical states it might be in, given the previous sequence of actions and percepts
  - is a set of physical states: the agent is in one of these states (but does not know in which one)
  - contains the actual physical state the agent is in
  - ex: $\{1, 2\}$: the agent is either in state 1 or in state 2 (but it does not know in which one)
- 2^n possible belief states out of n possible physical states!

Outline

Search with No Observation

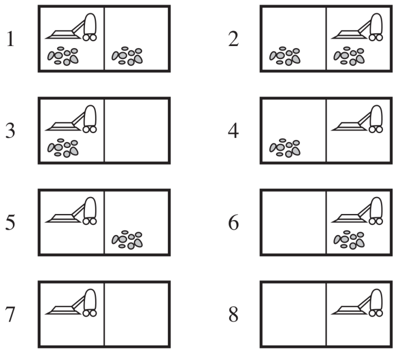
Search with No Observation

- aka Sensorless Search or Conformant Search
- Idea: To solve sensorless problems, the agent searches in the space of belief states rather than in that of physical states
  - fully observable, because the agent knows its own belief space
  - solutions are always sequences of actions (no contingency plan), because percepts are always empty and thus predictable
- Main drawback: 2^N candidate states rather than N

Search with No Observation: Example

Example: Sensorless Vacuum Cleaner

- the vacuum cleaner knows the geography of its world, but it doesn't know its location or the distribution of dirt
  - initial state: $\{1, 2, 3, 4, 5, 6, 7, 8\}$
  - after action RIGHT, state is $\{2, 4, 6, 8\}$
  - after action sequence [RIGHT,SUCK], state is $\{4, 8\}$
  - after action sequence [RIGHT,SUCK,LEFT,SUCK], state is $\{7\}$



Belief-State Problem Formulation

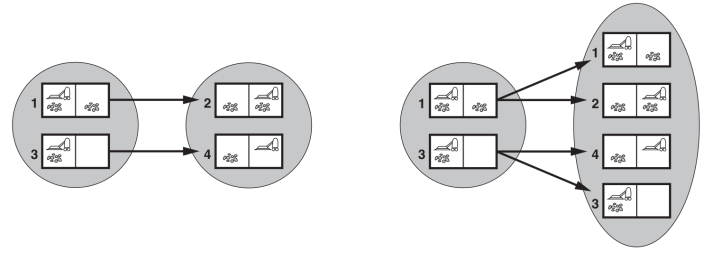
- Belief states: subsets of physical states
  - If P has N states, then the sensorless problem has up to 2^N states
- Initial state: typically the set of all physical states in P
- Actions: (assumption: illegal actions have no effects)
  - $Actions(b) \stackrel{\text{def}}{=} \bigcup_{s \in b} Actions_P(s)$
- Transition model:
  - for deterministic actions:
    $b' = Result(b, a) \stackrel{\text{def}}{=} \{s' \mid s' = Result_P(s, a) \text{ and } s \in b\}$
  - for nondeterministic actions: $b' = Result(b, a) \stackrel{\text{def}}{=}$
    $\{s' \mid s' \in Result_P(s, a) \text{ and } s \in b\} = \bigcup_{s \in b} Result_P(s, a)$
  - This step is called Prediction: $b' \stackrel{\text{def}}{=} Predict(b, a)$
- Goal test: $GoalTest(b)$ holds iff $GoalTest_P(s)$ holds, $\forall s \in b$
- Path cost: (assumption: cost of an action the same in all states)
  - $StepCost(a, b) \stackrel{\text{def}}{=} StepCost_P(a, s)$, $\forall s \in b$

$Actions_P()$, $Result_P()$, $GoalTest_P()$, $StepCost_P()$ refer to physical System P

Belief-State Problem Formulation [cont.]

Example: Sensorless Vacuum Cleaner, plain and slippery versions

Prediction: $\mathit{Result}(\{1,3\}, \mathit{Right})$, deterministic (a) and nondeterministic action (b)

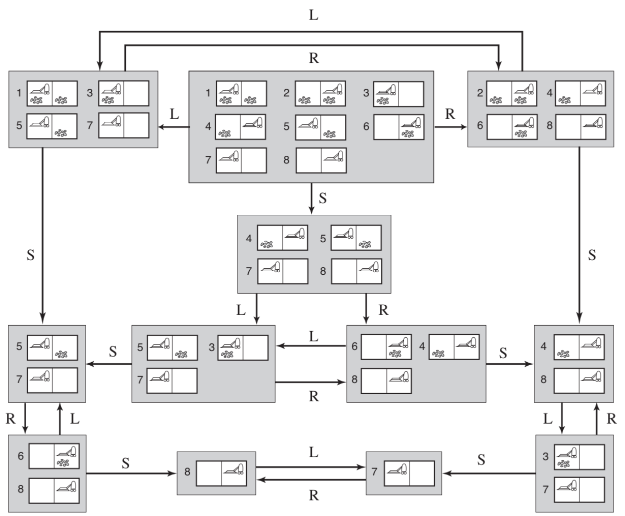


(a) (b)

Belief-State Problem Formulation [cont.]

Example: Sensorless Vacuum Cleaner: Belief State Space

(note: self-loops are omitted)



(© S. Russell & P. Norwig, AIMA)

Belief-State Problem Formulation [cont.]

Remarks

- if $b \subseteq b'$, then $Result(b, a) \subseteq Result(b', a)$
- If a is deterministic, then $|Result(b, a)| \leq |b|$
- The agent might achieve the goal earlier than $GoalTest(b)$ holds, but it does not know it

Properties

- An action sequence is a solution for b iff it leads b to to a goal
- If an action sequence is a solution for a belief state b, then it is also a solution for any belief state b' s.t. $b' \subseteq b$
  - if $b \overset{a_1}{\mapsto} .... \overset{a_k}{\mapsto} g$, then $b' \overset{a_1}{\mapsto} .... \overset{a_k}{\mapsto} g$

We can apply to the Belief-State space any search algorithm.

- we can discard a path reaching a belief state b if $b' \subseteq b$ has already been generated and discarded
- if a solution for b has been found, then any $b' \subseteq b$ is solvable

$\Longrightarrow$ Dramatically improves efficiency

Outline

Search with Observations

Perception and Belief-State Problem Formulation

- $Percept(s)$ returns the percept received in state s
  - if sensing is nondeterministic, it returns a set of possible percepts
  - ex: (local-sensing vacuum cleaner, can perceive dirty/clean only on the current position): $Percept(1) = [A, Dirty]$
  - with fully observable problems: $Percept(s) = s, \forall s$
  - with sensorless problems: $Percept(s) = null, \forall s$
- Partial observations: many states can produce the same percept
  - ex: $Percept(1) = Percept(3) = [A, Dirty]$
- $Actions()$, $StepCost()$, $GoalTest()$: as with sensorless case

Transition Model with Perceptions

- Three steps (aka prediction-observation-update process):
  1. Prediction: (same as for sensorless): $\hat{b} = Predict(b, a) \stackrel{\text{def}}{=} Result_{(sensorless)}(b, a) = \{s' \mid s' = Result_P(s, a) \text{ and } s \in b\}$
  2. Observation prediction: determines the set of percepts that could be observed in the predicted belief state
     $PossiblePercepts(\hat{b}) \stackrel{\text{def}}{=} \{o \mid o = Percept(s) \text{ and } s \in \hat{b}\}$
  3. Update: for each percept o, determine the belief state b_o, i.e., the set of states in $\hat{b}$ that could have produced the percept o
     - $b_o = Update(\hat{b}, o) \stackrel{\text{def}}{=} \{s \mid s \in \hat{b} \text{ and } o = Percept(s)\}$

$$\Longrightarrow Result(b,a) = \left\{ b_o \;\middle|\; \begin{array}{rl} b_o = & Update(Predict(b,a), o) \text{ and} \\ o \in & PossiblePercepts(Predict(b,a)) \end{array} \right\}$$

- set (not union!) of belief states, one for each possible percepts o
- $b_o \subseteq \hat{b}, \forall o \Longrightarrow$ sensing reduces uncertainty!
- (if sensing is deterministic) the b_o's are all disjoint
  $\Longrightarrow \hat{b}$ partitioned into smaller belief states, one for each possible next percept

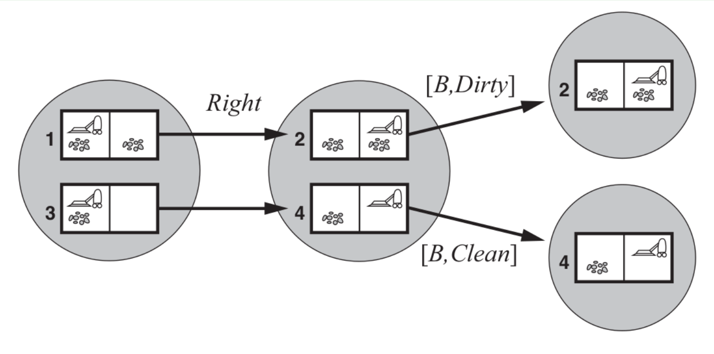
$\Longrightarrow$ Non-deterministic belief-state problem

- due to the inability to predict exactly the next percept

Transition Model with Perceptions: Example

Deterministic: Local-sensing vacuum cleaner

- $\hat{b} = Predict(\{1,3\}, Right) = \{2,4\}$
- $PossiblePercepts(\hat{b}) = \{[B, Dirty], [B, Clean]\}$
- $Result(\{1,3\}, Right) = \{\{2\},\{4\}\}$

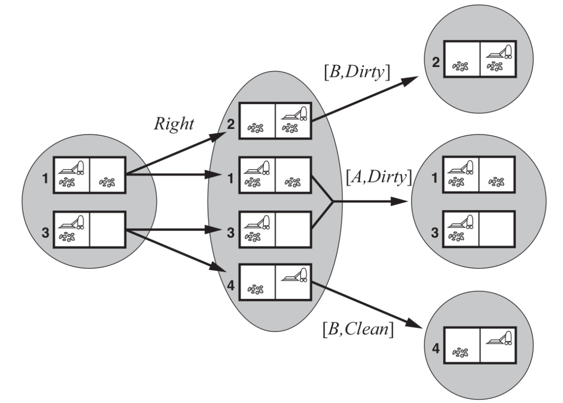


(© S. Russell & P. Norwig, AIMA)

Transition Model with Perceptions: Example

Nondeterministic: Slippery local-sensing vacuum cleaner

- $\hat{b} = \mathit{Predict}(\{1,3\}, \mathit{Right}) = \{1,2,3,4\}$
- $\mathit{PossiblePercepts}(\hat{b}) = \{[B, \mathit{Dirty}], [A, \mathit{Dirty}], [B, \mathit{Clean}]\}$
- $\mathit{Result}(\{1,3\}, \mathit{Right}) = \{\{2\}, \{1,3\}, \{4\}\}$



Solving Partially-Observable Problems

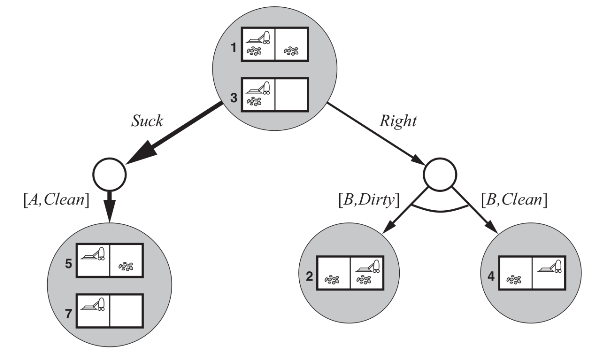
- Formulation as a nondeterministic belief-state search problem

⟹ The AND-OR search algorithms can be applied

⟹ The solution is a conditional plan

Solution for [A, Dirty]: [Suck, Right, if Bstate = 6 then Suck else [ ]]

First level:



(© S. Russell & P. Norwig, AIMA)

An Agent for Partially-Observable Environments

- Agent quite similar to the simple problem-solving agent [Ch.3]
  1. formulates a problem as a belief-state search
  2. calls an AND-OR-GRAPH search algorithm
  3. executes the solution
- Two main differences:
  - the solution is a conditional plan, not an action sequence
  - the agent needs to maintain its belief state as it performs actions and receives percepts (aka monitoring, filtering, state estimation)
- Similar to the prediction-observation-update process:
  - simpler, because the percept o is given by the environment $\Longrightarrow$ no need to calculate it
  - given b, a and o: $b' = Update(Predict(b, a), o)$

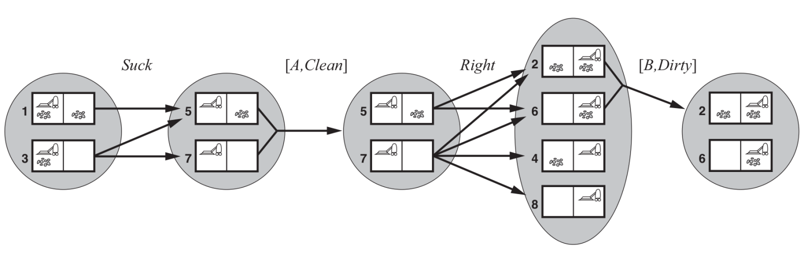
Remark

The computation has to happen as fast as percepts are coming in
$\Longrightarrow$ in some complex applications, compute approximate belief states

Example: Belief-State Maintenance

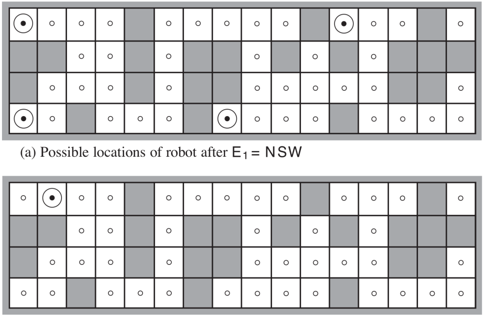
Example: Kindergarden Vacuum-Cleaner

- local sensing $\Longrightarrow$ partially observable
- any square may become dirty at any time unless the agent is actively cleaning it at that moment $\Longrightarrow$ nondeterministic
- Ex: $Update(\overbrace{Predict(\{1,3\}, Suck)}^{\{5,7\}}, [A, Clean]) = \{5,7\}$
- Ex: $Update(\overbrace{Predict(\{5,7\}, Right)}^{\{2,4,6,8\}}, [B, Dirty]) = \{2,6\}$



Example:

- Knows the map, senses walls in the four directions (NESW)
  - localization broken: does not know where it is
  - navigation broken: does not know the direction is moving to
  - goal: localization (know where it is)
- $b = \{all\ locations\}$, $o = NSW$
  1. $b_o = Update(b, NSW) = (a)$
  2. $b_o = Update(Predict(Update(b, NSW), Move), NS) = (b)$



(a) Possible locations of robot after $E_1 = NSW$

(b) Possible locations of robot After $E_1 = NSW, E_2 = NS$

Outline

Generalities

Online vs. offline search

- So far: Offline search
  - it computes a complete solutions before executing it
- Online search: agent interleaves computation and action
  - it takes an action,
  - then it observes the environment and computes the next action
  - (repeat)
- Useful in nondeterministic domains
  - prevents search blowup
- Necessary in dynamic domains or unknown domains
  - cannot know the states and consequences of actions
  - faces an exploration problem: must use actions as experiments in order to learn enough
  - ex: a robot placed in a new building
    $\Longrightarrow$ must explore it to build a map to use for getting from A to B
  - ex: newborn baby $\Longrightarrow$ acts to learn the outcome of his/her actions

Must be solved by executing actions, rather than by pure computation

Online Search

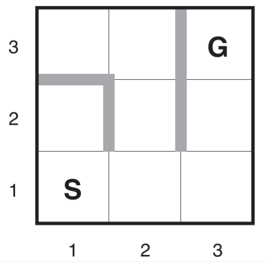
Working Hypotheses

- Assumption: a deterministic and fully observable environment
- The agent knows only
  - *Actions*(s), which returns the list of actions allowed in s
  - the step-cost function $c(s, a, s')$ (cannot be used until s' is known)
  - *GoalTest*(s)
- Remark: The agent cannot determine *Result*(s, a)
  - except by actually being in s and doing a
- The agent knows an admissible heuristic function h(s), that estimates the distance from the current state to a goal state
- Objective: reach goal with minimal cost
  - Cost: total cost of traveled path
  - Competitive ratio: ratio of cost over cost of the solution path if search space is known ($+\infty$ if agent in a deadend)

Online Search: Example

Example: a simple maze problem

- the agent does not know that going Up from (1,1) leads to (1,2)
- having done that, it does not know that going Down leads to (1,1)
- the agent might know the location of the goal
- it may be able to use the Manhattan-distance heuristic



Online Search: Deadends

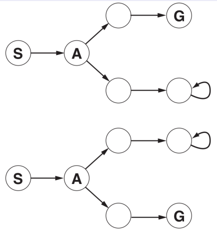
Inevitability of Deadends

- Online search may face deadends (e.g., with irreversible actions)
- No algorithm can avoid dead ends in all state spaces
- Adversary argument: for each algo, an adversary can construct the state space while the agent explores it
  - If states S and A visit. What next?

$\Longrightarrow$ if algo goes right, adversary builds (top), otherwise builds (bot)

$\Longrightarrow$ adversary builds

- Assumption the state space is safely explorable: some goal state is reachable from every reachable state (ex: reversible actions)



Online Search Agents

Online Search Agents: Basic Ideas

- Idea: The agent creates & maintains a map of the environment ($result[s, a]$)
  - map is updated based on percept input after every action
  - map is used to decide next action
- Difference wrt. offline algorithms (ex A^*, BFS)
  - Can only expand the node it is physically in

    $\Longrightarrow$ expand nodes in local order

    $\Longrightarrow$ DFS natural candidate for an online version
  - Needs to backtrack physically
    - DFS: go back to the state from which the agent most recently entered the current state
    - must keep a table with the predecessor states of each state to which the agent has not yet backtracked ($unbacktracked[s]$)

    $\Longrightarrow$ backtrack physically (find an action reversing the generation of s)
  - Works only if actions are always reversible
  - Worst case: each node is visited twice
  - An agent can go on a long walk even if it is close to the solution
    - an online iterative deepening approach solves this problem

Online Search Agents [cont.]

function ONLINE-DFS-AGENT(s') **returns** an action
  inputs: s', a percept that identifies the current state
  persistent: *result*, a table indexed by state and action, initially empty
    untried, a table that lists, for each state, the actions not yet tried
    unbacktracked, a table that lists, for each state, the backtracks not yet tried
    s, a, the previous state and action, initially null

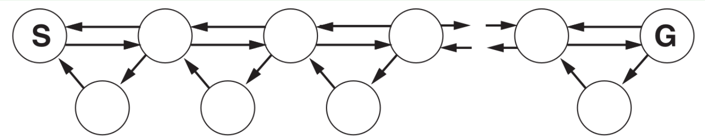
  if GOAL-TEST(s') **then return** *stop*
  if s' is a new state (not in *untried*) **then** *untried*[s'] ← ACTIONS(s')
  if s is not null **then**
    result[s, a] ← s'
    add s to the front of *unbacktracked*[s']
  if *untried*[s'] is empty **then**
    if *unbacktracked*[s'] is empty **then return** *stop*
    else a ← an action b such that *result*[s', b] = POP(*unbacktracked*[s'])
  else a ← POP(*untried*[s'])
  $s \leftarrow s'$
  return a

Online Local Search

- Hill Climbing natural candidate for online search
  - locality of search
  - only one state is stored
  - unfortunately, stuck in local minima
  - random restarts not possible
- Possible solution: Random Walk
  - selects randomly one available actions from the current state
  - preference can be given to actions that have not yet been tried
  - eventually finds a goal or complete its exploration if space is finite
  - unfortunately, very slow

Random Walk: example

- random walk takes exponentially many steps to find a goal (backward progress is twice as likely as forward progress)



(© S. Russell & P. Norwig, AIMA)

Online A^*: $LRTA^*$

$LRTA^*$: General ideas

- Better possible solution: add memory to hill climbing
- Idea: store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited
  - initially $h(s)$
  - updated as the agent gains experience in the state space

  (recall that $h(s)$ is in general "too optimistic")

$\Longrightarrow$ Learning Real-Time A^* ($LRTA^*$)

- builds a map of the environment in the result[s,a] table
- chooses the "apparently best" move a according to current $H()$
- updates the cost estimate $H(s)$ for the state s it has just left, using the cost estimate of the target state s'
  - $H(s) := c(s, a, s') + H(s')$
- "optimism under uncertainty": untried actions in s are assumed to lead immediately to the goal with the least possible cost $h(s)$

$\Longrightarrow$ encourages the agent to explore new, possibly promising paths

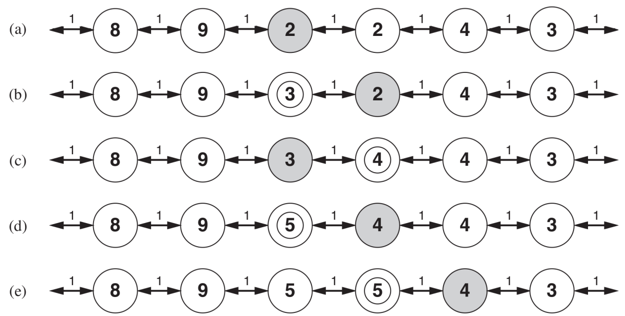
Online A^*: $LRTA^*$

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
              H, a table of cost estimates indexed by state, initially empty
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null
     result[s, a] ← s'
     H[s] ←       min        LRTA*-COST(s, b, result[s, b], H)
            b ∈ ACTIONS(s)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

Example: *LRTA**

Five iterations of *LRTA** on a one-dimensional state space

- states labeled with current H(s), arcs labeled with step cost
- shaded state marks the location of the agent,
- updated cost estimates a each iteration are circled



(© S. Russell & P. Norwig, AIMA)