EBERHARD KARLS UNIVERSITÄT TÜBINGEN

MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Artificial Intelligence
Chapter 4: Beyond Classical Search

Andreas Zell

After the Textbook: Artificial Intelligence,
A Modern Approach
by Stuart Russel and Peter Norvig (3rd Edition)

4. Beyond Classical Search

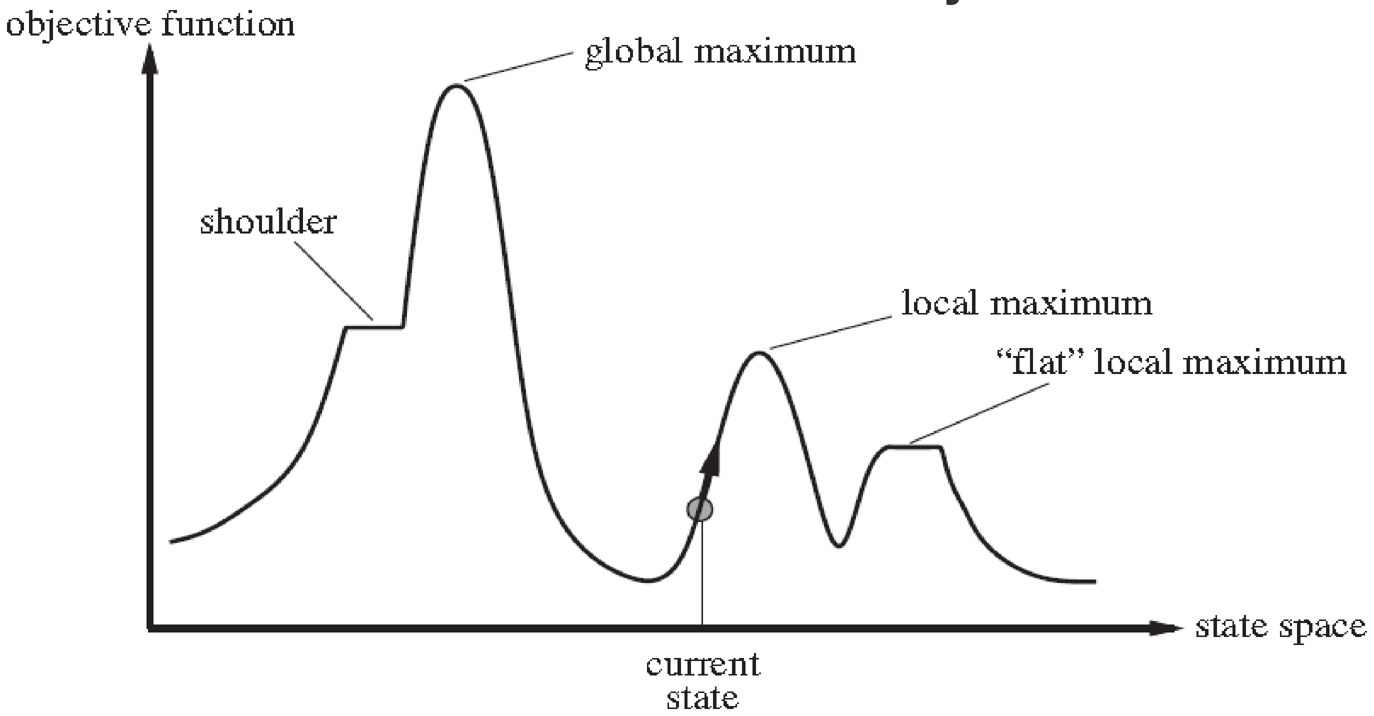
- Search algorithms considered so far
  - explore the search space systematically
  - keep one or more paths in memory
  - record which alternatives have been explored at each point along the path
- For many problems the path is irrelevant and only the configuration of the goal state is important
- Local search algorithms
  - Store only the current node
  - Generally move only to neighbors of this node
  - Do not retain the path to this node

4.1 Local Search Algorithms

- Local search algorithms
  - Are not systematic
  - Require little memory – usually only a constant amount to store a set of current states
  - Often find reasonable solutions in large or even infinite search spaces, where systematic algorithms fail
  - Can be used on optimization problems
- Optimization problems do not fit the standard model from chapter 3 and often have no goal test or path cost

4.1 State-Space Landscape

- Local search algorithms work on state-space landscapes which are defined by the relationship between “location” (state) and “elevation” (value of heuristic cost function or objective function)



4.1 State-Space Landscape

- If “elevation” corresponds to cost, the aim is finding the lowest valley – the global minimum
- If “elevation” corresponds to an objective function, the aim is finding the highest peak – the global maximum
- Conversion between both types can be easily done by inserting a minus sign
- A local search algorithm is called complete if it always finds a goal if one exists and is called optimal if the found goal is a global minimum/maximum

4.1.1 Hill Climbing Search

- Basic idea of hill-climbing search (HC):
  - continuously move to neighbor states of increasing value and terminate at a “peak” if no better neighbor than the current state exists
- Only current state with its objective value is stored
- HC does not look beyond immediate neighbors of the current state

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

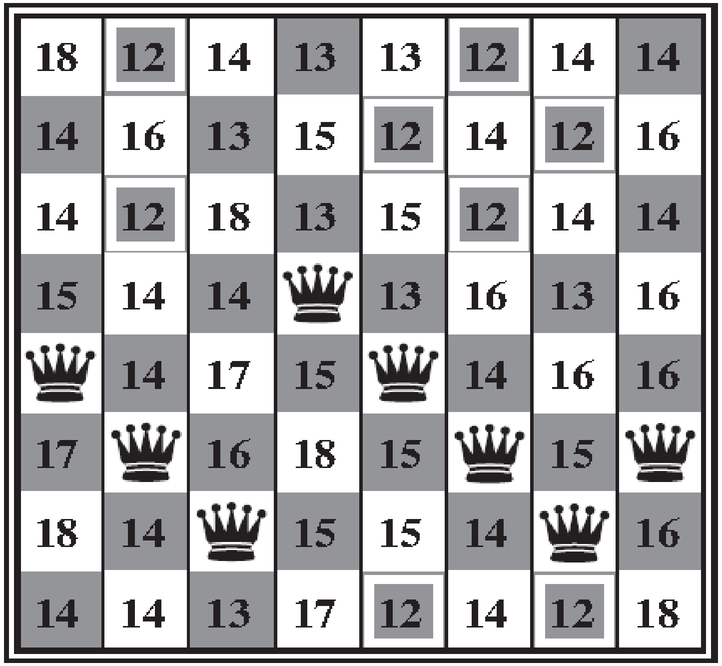
$current \leftarrow$ MAKE-NODE($problem$.INITIAL-STATE)
loop do
  $neighbor \leftarrow$ a highest-valued successor of $current$
  if neighbor.VALUE $\leq$ current.VALUE **then return** $current$.STATE
  $current \leftarrow neighbor$

4.1.1 Hill-Climbing Example 8-Queens

- Local search algorithms typically use a complete state formulation
- In case of the 8-queen problem each state has 8 queens on the board and stores the row of the queen for each column
- Successor states are all possible states that can be generated by moving a single queen on the board to another row on the same column
- Cost function h is the number of pairs of queens that are attacking each other
- Global minimum is a state with h=0

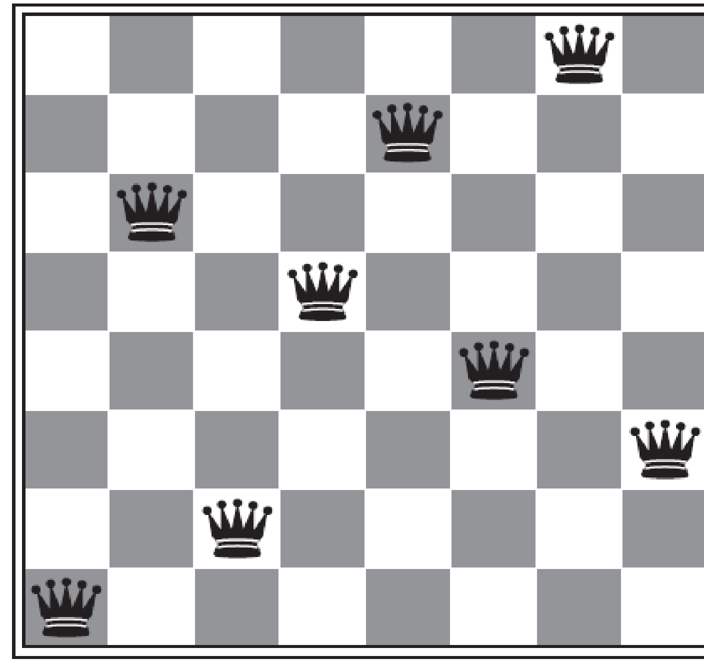
4.1.1 Hill-Climbing Example 8-Queens

EBERHARD KARLS UNIVERSITÄT TÜBINGEN



state (4,3,2,5,4,3,2,3) *h*=17

with costs for the successors obtained by moving the queen of this column to this row



local minimum with *h*=1 each possible successor has a higher or equal cost

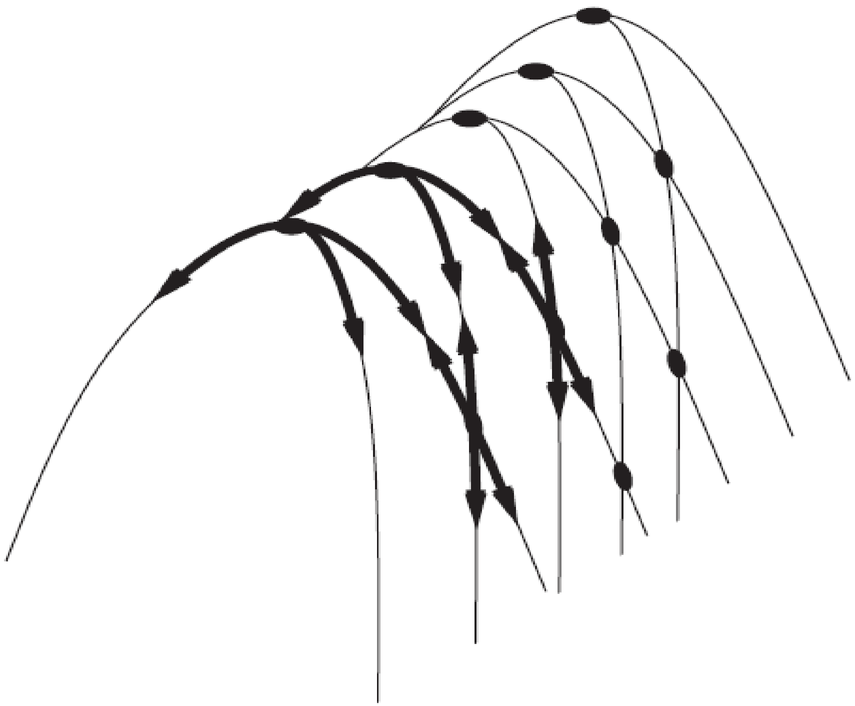
4.1.1 Greedy Local Search

- Hill-climbing is a greedy local search since it always moves to the best neighbor state without considering the next moves
- Greedy strategies often make rapid progress at the start but risk to get stuck
- Reasons for getting stuck
  - Local maxima: all neighbor states have a worse value than the current state, so no move possible
  - Plateaux: neighbor states have same value as current state; no sense of direction; hill climbing might get lost

    can be alleviated by allowing sideway moves but may end in a loop, therefore requires stopping criterion

4.1.1 Greedy Local Search

- Reasons for getting stuck:
  - Ridges: Sequence of local maxima that are connected by states with lower value



4.1.1 Hill-Climbing Variants

- Stochastic hill climbing: chooses random uphill moves instead of the best; slower convergence but finds better solutions in some landscapes
- First-choice hill climbing: instead of generating all moves, subsequently generate moves until a state is found which is better than the current one
  - good strategy for landscapes with extremely high number of neighbor states
- Random-restart hill climbing: Repeat hill-climbing runs starting from random initial states until goal is found; trivially complete since an initial state which is also goal may eventually be generated

4.1.2 Simulated Annealing

- Simulated annealing (SA) works similar to hill-climbing but also allows “downhill” moves
- Mimicks the annealing process in metallurgy, which starts at high temperature T and allows to reach a low energy state by gradual cooling
- A random move is generated at each iteration
- If it improves the objective function the move is always accepted
- If the objective function is worsened the move is accepted with probability p
- p is exponentially reduced with the “badness” of the move and also decreases with the temperature T

4.1.2 Simulated Annealing

- Simulated annealing starts with high *T* which is reduced over time; bad moves are more likely to be allowed at the start and become more unlikely
- If *T* is lowered slowly enough SA finds the global optimum with probability 1

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

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

4.1.3 Local Beam Search

- Instead of only one state local beam search keeps track of *k* states in parallel
- It is initialized with *k* random states and at each step generates all possible successors of the *k* current states
- If one of the successors is the goal, local beam search stops; otherwise the *k* best successors are selected as the new current states
- Can suffer from lack of diversity, since the *k* states quickly become concentrated at a small region turning local beam search into a form of hill-climbing
- Stochastic beam search alleviates this by randomly selecting the *k* successors with a probability corresponding to their objective function values

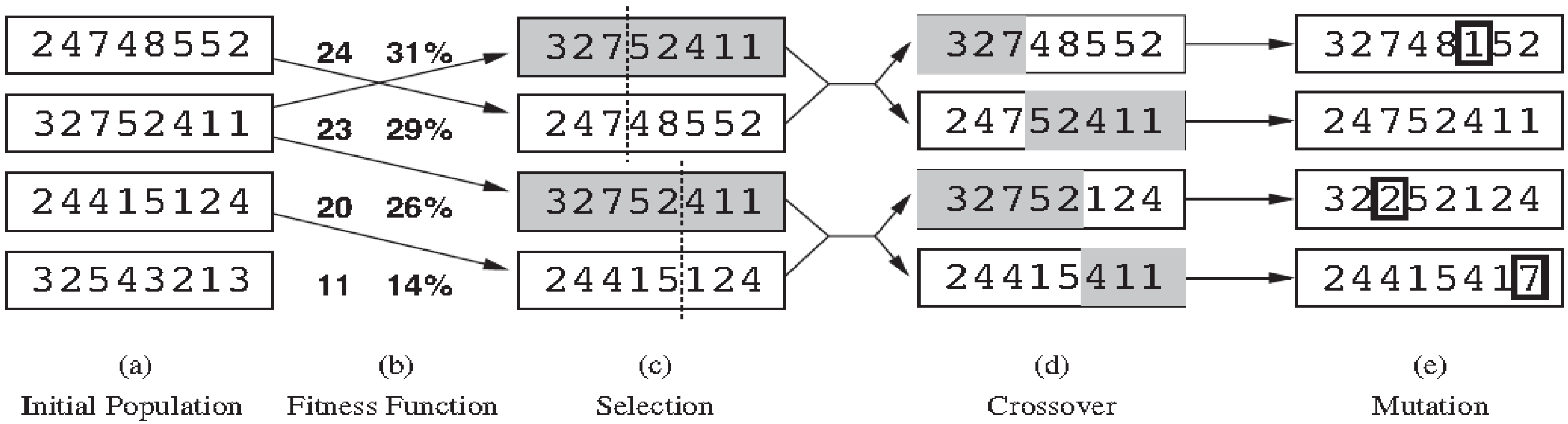
4.1.4 Genetic Algorithms

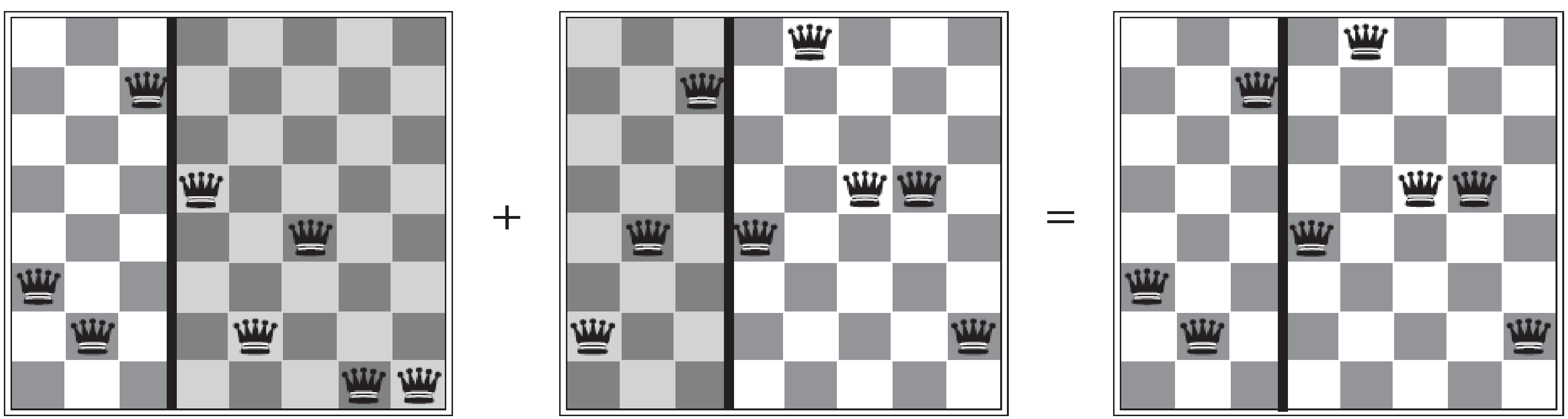
- Genetic algorithms are a variant of stochastic beam search based on principles of natural evolution
- States are represented by individuals within a population
- Each iteration new individuals are generated by crossover of two randomly selected parents within the current population
- The probability to be selected as parent corresponds to the objective function value respectively fitness function value of the individual
- With a low probability the individuals are slightly changed by mutation after crossover
- Lecture in summer semester "Evolutionäre Algorithmen"

```
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))
```

4.1.4 Genetic Algorithms Example





4.2 Local Search in Continuous Spaces

- Almost all algorithms (with exception of hill-climbing and simulated annealing) discussed so far can not handle continuous search spaces
- Since the development of calculus by Newton and Leibniz in the 17th century numerous algorithms for continuous optimization have been published
- As example consider the problem of placing three airports in Romania, so that the sum of the squared distances of these airports to their closest cities is minimal
- The state space is then defined by the coordinates of the airports: $(x_1,y_1),(x_2,y_2),(x_3,y_3)$

- The resulting search space contains 6 variables
- Moving around in the search space corresponds to moving one or more airports
- Let C_i be the set of closest cities for airport *i*, then the objective function for the neighborhood of the current state, where C_i remains constant, is:

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} \left((x_i - x_c)^2 + (y_i - y_c)^2 \right)$$

- This expression is correct locally but not globally because the relationship of a city to the sets C_i is a discontinuous function of the state

- A way to avoid continuous problems is to discretize the neighborhood of each state
- In the example this can be done by moving only one airport by a fixed amount $\pm\delta$ giving 12 possible successors for each state
- The resulting problem can be solved by any previously described local search algorithm
- Simulated annealing and stochastic hill climbing can also be applied with randomly generated vectors of length δ

4.2 Using gradient information

- Many methods attempt to navigate in the landscape using the gradient which is the magnitude and direction of the steepest slope
- In our example the gradient is:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right)$$

- Some problems can be solved by solving the equation $\nabla f = 0$
- In many cases this equation can not be solved in closed form, as in our example, since the gradient depends on C_i which is discontinuous

4.2 Using gradient information

- We can still compute the gradient locally for example for airport 1 and constant C_1

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_i - x_c)$$

- Given a locally correct gradient we can perform steepest ascent hill-climbing by updating the state according to

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$$

- α is the step size, which is a small constant
- If f is not differentiable, an empirical gradient can be obtained by evaluating the response to small changes in each variable

4.2 Line Search

- Determining optimal α is complicated. Too small α leads to many too small steps. Too large α leads to overshooting the optimum
- Line search tries to overcome this by extending the current gradient direction until f starts to worsen again
- For many problems the most efficient method is the Newton-Raphson algorithm, a general technique for finding roots of functions, solving the equation $g(x)=0$ by iteratively computing new estimates for the root x by:

$$x \leftarrow x - g(x) / g'(x)$$

4.2 Newton-Raphson algorithm

- For optimizing f we need to find an $\mathbf{x}$ so that $\nabla f=0$
- By setting $\nabla f=g(\mathbf{x})$ we obtain:

$$\mathbf{x} \leftarrow \mathbf{x} - H_f^{-1}(\mathbf{x}) \cdot \nabla f(\mathbf{x})$$

- $\boldsymbol{H}_f(\mathbf{x})$ is the Hessian matrix of second derivatives with the elements $\partial^2 f / \partial x_i \partial x_j$
- For the airport example $\boldsymbol{H}_f(\mathbf{x})$ is zero for all off-diagonal elements and the diagonal elements for airport i are twice the number of cities in C_i
- An update step then moves the airport i directly to the centroid of C_i

4.2 Newton-Raphson algorithm

- For high-dimensional problems computing the n^2 elements of $\boldsymbol{H}_f(\mathbf{x})$ is expensive, therefore many approximate versions of the method have been developed
- As local search algorithm gradient based algorithms also suffer from local optima, ridges, plateaux
- Random restart or simulated annealing can still be used

4.2 Constrained Optimization

- A final topic to this is constrained optimization, which are problems whose solutions must satisfy some hard constraints on the values of the variables
- The airports in the example might be constrained to be inside Romania and on dry land
- The difficulty of such problems depends on the nature of the constraints
- Linear programming is the best known category, in which constraints must be linear inequalities forming a convex set and the objective must also be linear. Problems of this type can be solved in polynomial time on the number of variables

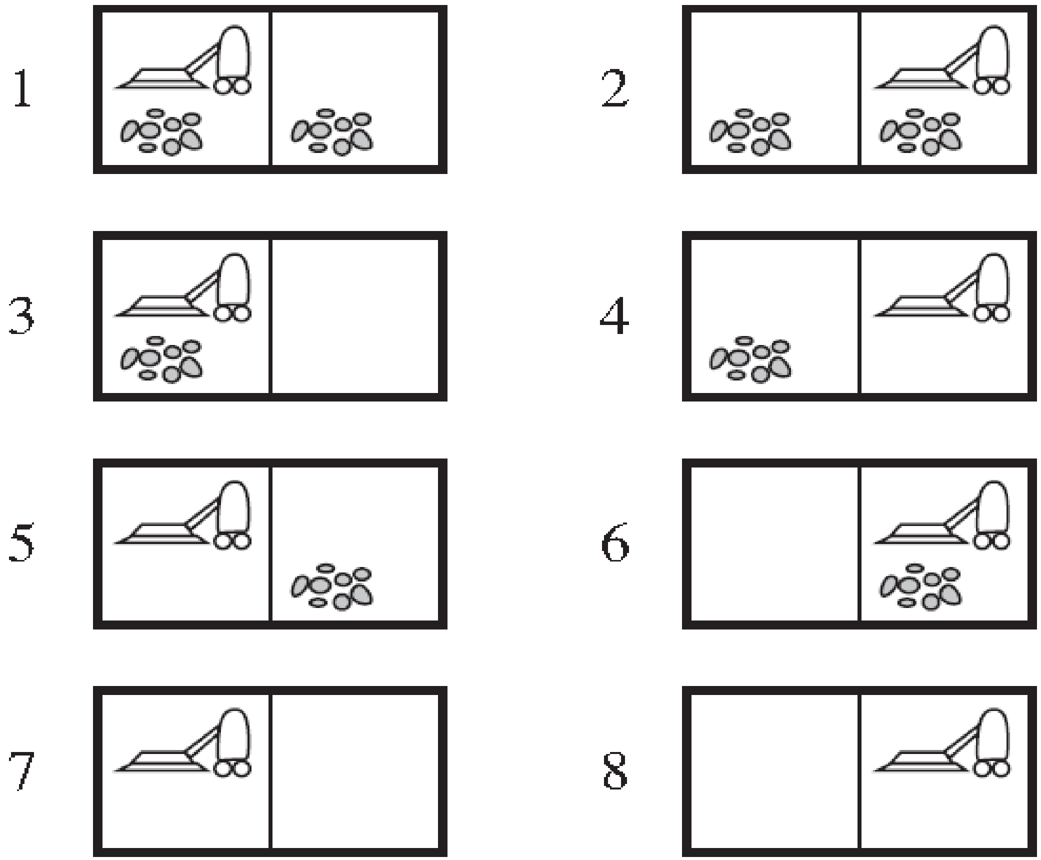
- So far we assumed that the environment is fully observable and that the agent knows the effect of each action
- When the environment is either partially observable or nondeterministic, percepts become useful
- In partially observable environments percepts help to narrow down the set of possible states the agent might be in
- In nondeterministic environments, percepts tell the agent which possible outcome of the action has actually occurred

4.3 Search with Nondeterministic Actions

- Percepts can not be determined in advance but the agents actions will depend on future percepts
- Thus the solution to a problem is not a sequence of actions but a contingency plan (strategy)
- As example we will use the vacuum cleaner world with the actions *Left, Right, Suck*
- In the erratic vacuum world *Suck* is redefined as:
  - When applied to a dirty square, the square is cleaned and an adjacent square sometimes is cleaned up, too.
  - When applied to a clean square, sometimes dirt is deposited on that square

4.3 Erratic Vacuum World

- The eight possible states of the erratic vacuum world – states 7 and 8 are goal states



4.3 Erratic Vacuum World

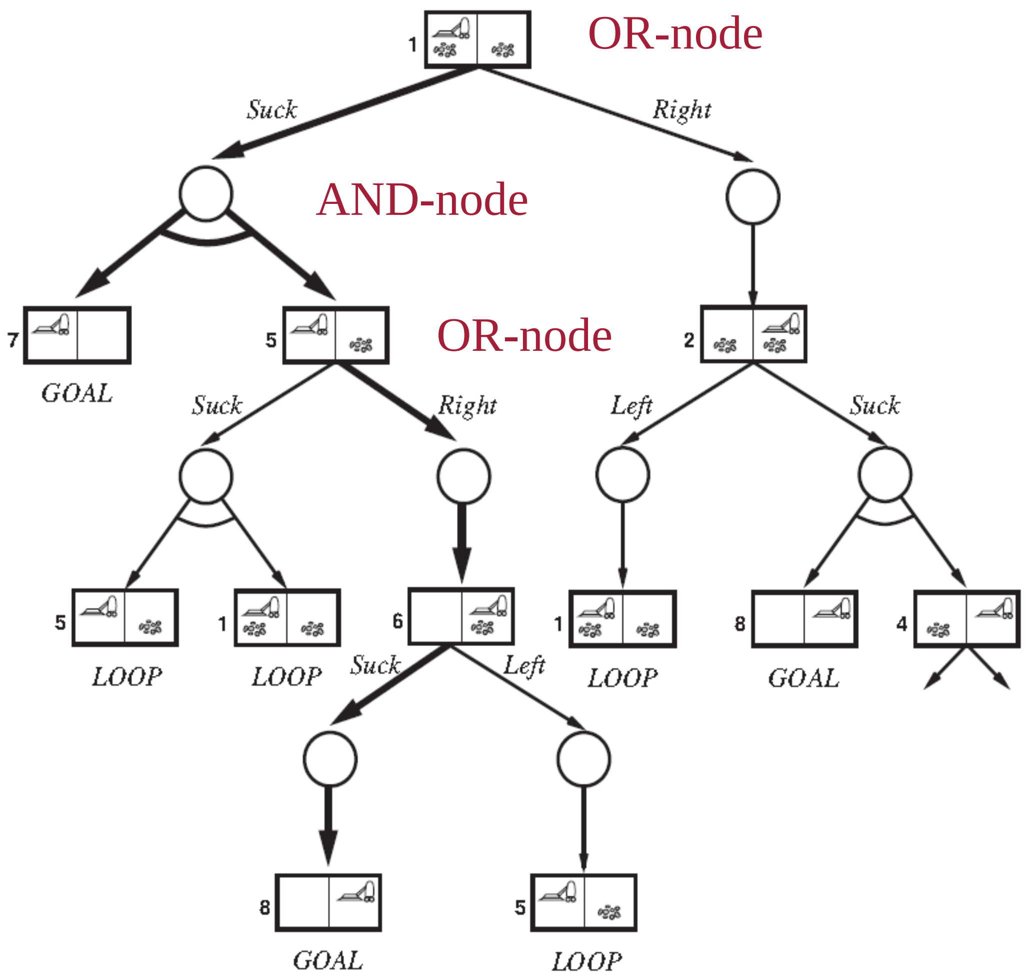
- To provide a precise formulation we need to generalize the transition model of Chapter 3
- Instead of defining the transition model by a RESULT function which returns a single state we use a RESULT function which returns a set of possible outcomes
- For example *Suck* in state 1 returns the set {5,7}
- We also need to generalize the concept of solution, since, for example, if we start in state 1 there is no single sequence of actions to solve the problem; instead we need a contingency plan like:

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

4.3 AND-OR Search Trees

- Thus solutions for nondeterministic problems can contain nested **if-then-else** statements
- This results in trees rather than sequences allowing to select actions based on contingencies arising during execution
- Many problems in the real, physical world are such contingency problems, since exact prediction is impossible
- In a deterministic environment branches in a tree search are introduced by the agents' own choices in each state. We call such nodes OR nodes

- In a nondeterministic environment branching is also introduced by the environment's choice of the outcome of each action. We call these AND nodes
- The two kind of nodes alternate leading to an AND-OR tree
- The bold path corresponds to the plan given before



4.3 AND-OR Search Trees

- A solution for an AND-OR search problem is a subtree that:
  1. has a goal node at every leaf
  2. specifies one action at each of its OR nodes
  3. includes every outcome branch at each of its AND nodes
- The resulting plan uses if-then-else notation to handle AND branches
- If there are more than two branches at a node a case construct might be better

- Modifying the basic problem-solving agent from 3.1 to execute such solutions is straightforward
- One might also consider a somewhat different agent design which acts before it has found a guaranteed plan and deals with contingencies as they arise
- This type of interleaving search and execution is also useful for exploration problems
- One key aspect is the way to deal with cycles which often arise in nondeterministic problems
- The following algorithm returns failure if the current state is identical to a state on the path from the root

- This does not mean there is no solution, only that if there is a non-cyclic solution it must be reachable from an earlier state

```
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
```

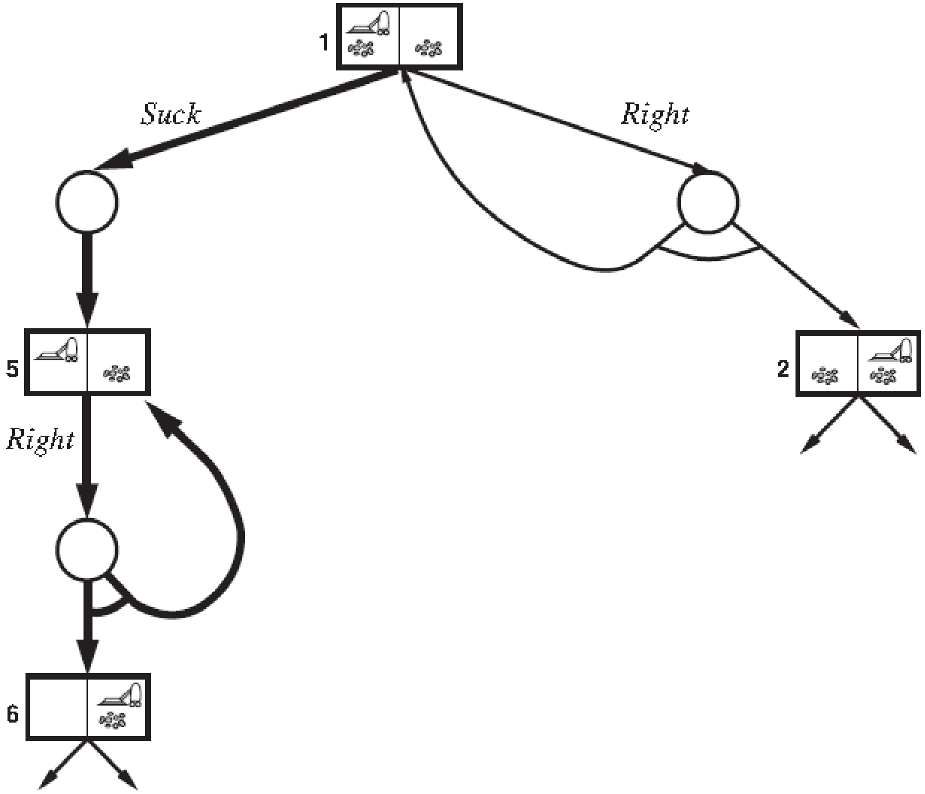
```
function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
   for each s_i in states do
       plan_i ← OR-SEARCH(s_i, problem, path)
       if plan_i = failure then return failure
   return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else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

4.3 AND-OR Search Trees

- AND-OR graphs can also be explored by breadth-first or best-first methods
- For this the concept of heuristic functions must be modified to estimate the cost of a contingent solution rather than a sequence
- The notion of admissibility carries over and there is an analog version of the A* algorithm
- Now consider the slippery vacuum world which is identical to the erratic vacuum world except that movement actions may fail, e.g. *Right* in the state 1 thus leads to the state set {1,2}

4.3 AND-OR Search Trees

- There are no longer any acyclic solutions from state 1
- AND-OR-graph-search would return failure
- However there is a cyclic solution which is to keep trying *Right* until it works



4.3 AND-OR Search Trees

- We can express this solution by adding a label denoting some portion of the plan and use it later

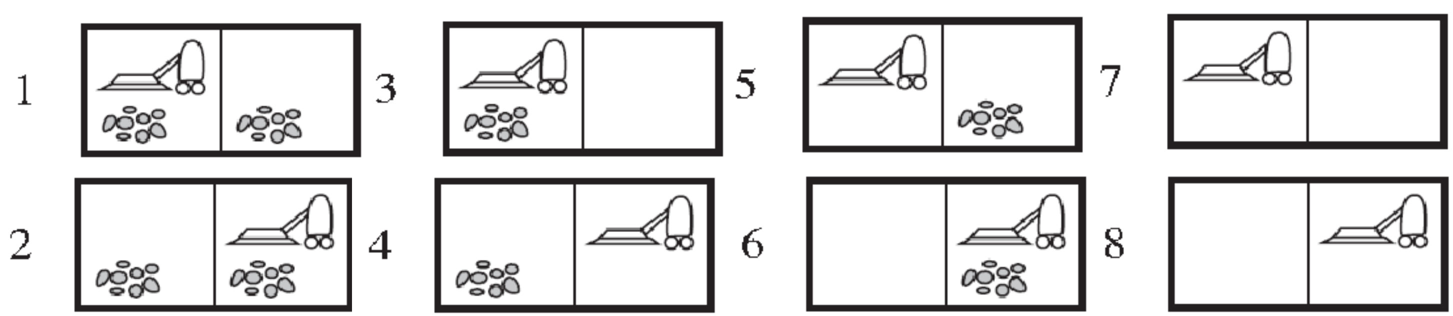
  [*Suck, L1 : Right* **if** *State*=5 **then** *L1* **else** *Suck*]
- A cyclic plan may be considered a solution provided that every leaf is a goal state and that a leaf is reachable from every point in the plan
- Given the definition of a cyclic solution an agent executing this solution will only eventually reach the goal with probability 1, if the nondeterminism is caused by a stochastic process, and not by a hidden property of the environment which prevents some state transitions

4.4 Searching with Partial Observations

- In problems with partial observability, the agent's percepts are insufficient to pin down the exact state
- An action may in this case lead to one of several states even if the environment is deterministic
- The key concept for such problems is the belief state which represents the agent's current belief about possible physical states it might be in, given the sequence of actions and percepts
- First we consider a sensorless agent, whose percepts provide no information at all
- Such problems with sensorless agents, are also called conformant problems

4.4.1 Searching with no Observation

- Sensorless Agents can still often act successful and actually be advantageous since they don’t rely on costly and potentially unreliable sensor information
- As example we assume a sensorless vacuum world where the agent knows the geography of its world but not its position or the distribution of dirt
- The initial state then can be any element of the set {1,2,3,4,5,6,7,8}

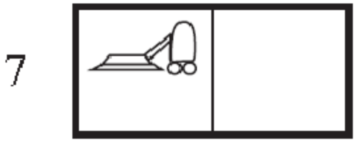


4.4.1 Searching with no Observation

- If we move to the *Right* we reduce the set of possible states to {2,4,6,8}

- Using *Suck* we can reduce the set further to {4,8}

- Using the sequence [*Right,Suck,Left,Suck*] guarantees to reach in the goal state 7 regardless of the initial state



4.4.1 Searching with no Observation

- For this problem we say the agent can coerce the world into state 7.
- To solve sensorless problems we search in the space of belief states rather than in the space of physical states.
- The belief space of the problem is fully observable because the agent always knows its belief state.
- The solution, if it exists, is always a sequence of actions, since the percepts are always empty and therefore completely predictable, so there are no contingencies to plan for.

4.4.1 Searching with no Observation

- For the physical problem *P* we can now define the corresponding sensorless problem as follows:
  - Belief states: the entire belief state space containing every possible set of physical states *P* (up to 2^N belief states if *P* has *N* possible states)
  - Initial state: Typically the set of all states in *P*
  - Actions: The actions in a belief state are the union of the actions of the physical states

    $$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s)$$

    (we here ignore forbidden actions)

4.4.1 Searching with no Observation

- Transition model:
  - The result of an action is the set of all physical states resulting from performing the action on all physical states in the current belief state
  - For deterministic actions the set is:

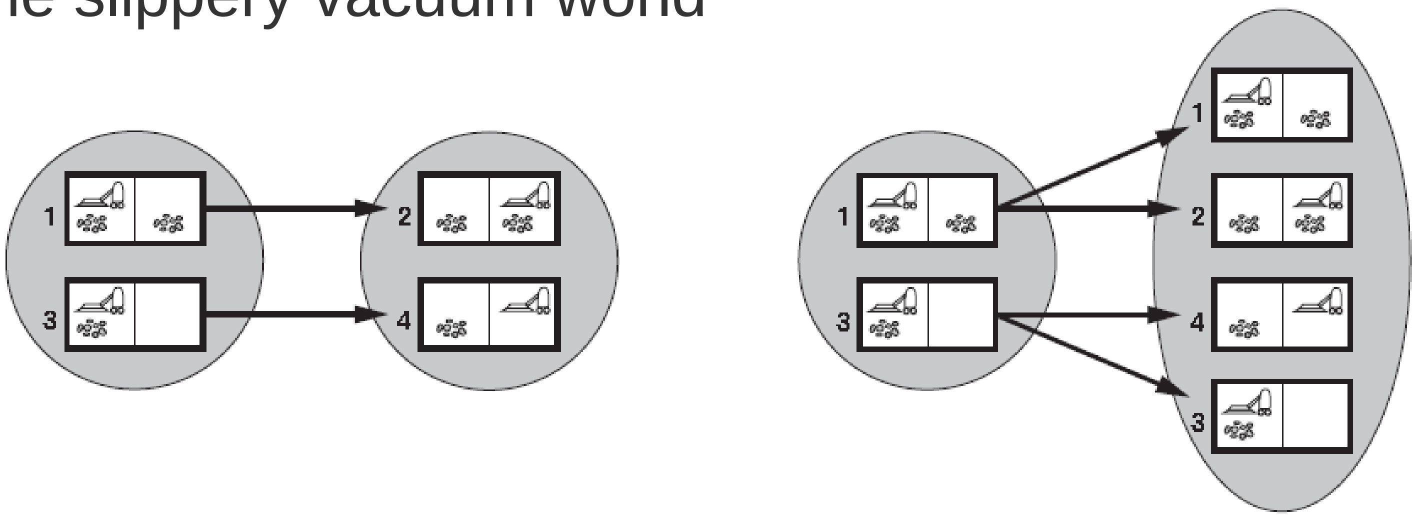
$$b' = \text{RESULT}(b,a) = \{s' : s' = \text{RESULT}_P(s,a) \text{ and } s \in b\}$$

  - For nondeterministic actions the set is:

$$b' = \text{RESULT}(b,a) = \{s' : s' \in \text{RESULT}_P(s,a) \text{ and } s \in b\}$$
$$= \bigcup_{s \in b} \text{RESULT}_P(s,a)$$

  - The process of generating a new belief state after action is called the prediction step $\text{PREDICT}_P(b,a)$

- For deterministic actions *b*' is never larger than *b*, but for nondeterministic actions *b*' might be larger
- (a) Predicting the next belief state for the sensorless vacuum world with the deterministic action *RIGHT*
- (b) Predicting the same belief state for the same action in the slippery vacuum world

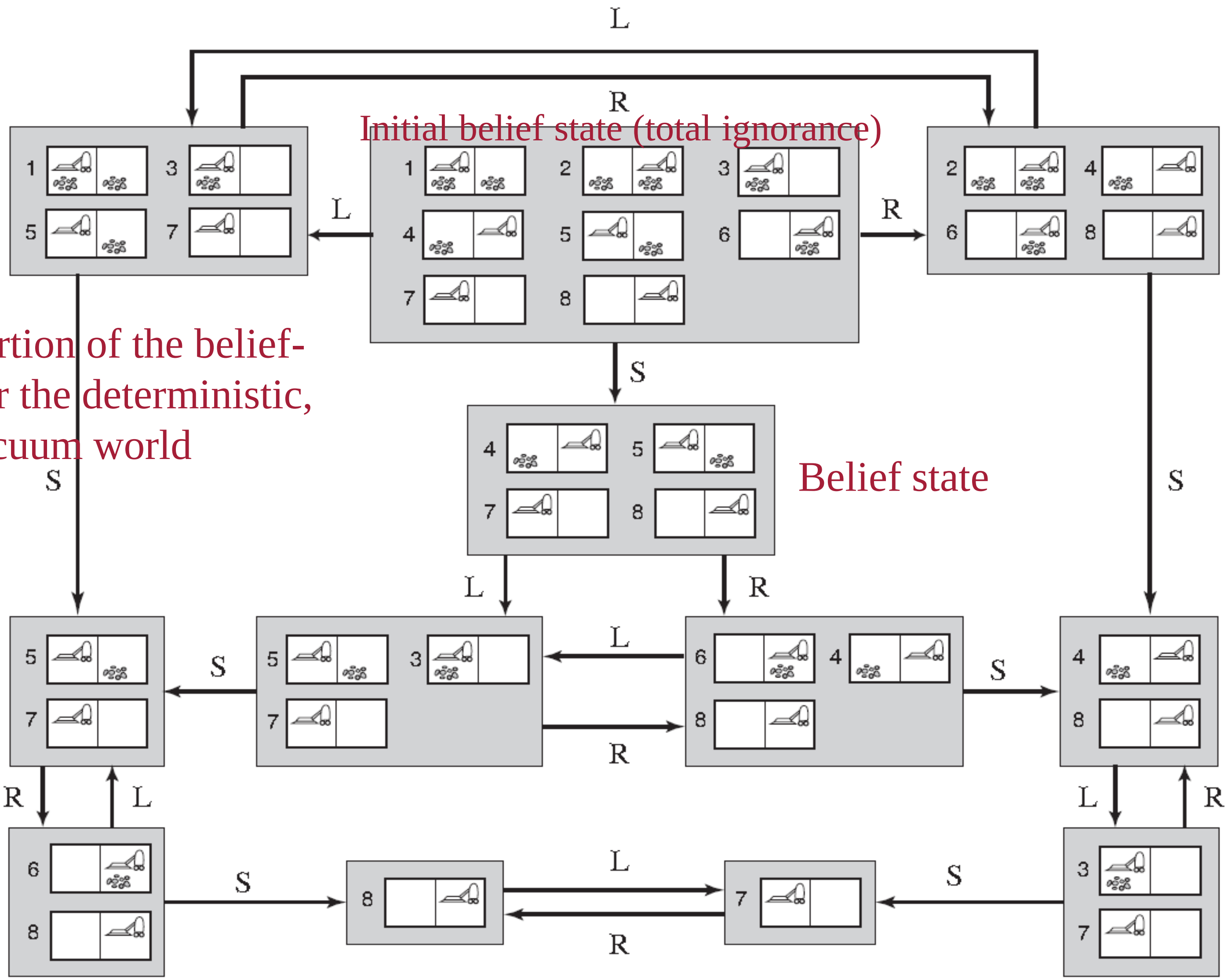


(a) (b)

4.4.1 Searching with no Observation

- Goal test:
  - Goal states in the belief space are only states where all physical states satisfy the GOAL-TEST$_P$
  - The agent may accidentally arrive at the goal earlier but it will not know this
- Path cost:
  - We assume that any action has identical cost for all physical states in the same belief state
- The definitions allow an automatic construction of the belief state problem
- Any previously described search algorithm can be applied

4.4.1 Searching with no observation



Reachable portion of the belief-state space for the deterministic, sensorless vacuum world

4.4.1 Searching with no Observation

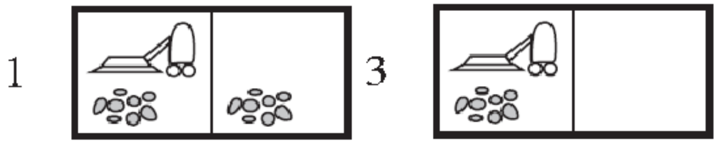
- Even with pruning, sensorless problem solving is seldom feasible using the algorithms described so far because of the size of the belief space
- For example the initial belief state of a 10x10 vacuum world contains 100×2^{100} or 10^{32} physical states
- A solution to this is to represent the belief state in a more compact description using formal representation schemes (Chapter 7)
- Another solution is to avoid treating belief states as black boxes
- Instead we can use incremental belief state search which looks inside belief states

4.4.2 Searching with Observations

- For general partially observable problems we have to specify how the environment generates percepts for the agent
- Using a PERCEPT(s) function we can extend the method to general partial observable problems
- PERCEPT(s) returns the percept of the agent for a given state s
- PERCEPT(s) returns a set of possible percepts for cases with nondeterministic sensing
- Fully observable problems have PERCEPT(s) =s
- Sensorless problems have PERCEPT(s) =*null*

4.4.2 Searching with Observations

- For partially observable problems the same percept could have been produced by several states, e.g. the percept [*A*,*Dirty*] can be produced by the states {1,3} in the vacuum world.



- We can keep ACTION, STEP-COST, and GOAL-TEST from the underlying problem just as for sensorless problems
- We have to incorporate the percepts into the transition model for the belief states

4.4.2 Searching with Observations

- We divide the transition from one belief state to an other into three stages: prediction, observation-prediction, update
- Prediction: is the same as for sensorless problems; given an action *a* in belief state *b* the predicted belief state is *b'*=PREDICT(*b*,*a*)
- Observation Prediction: determines the set of percepts *o* that could be observed in *b'*:

$$\text{POSSIBLE} - \text{PERCEPTS}(b') = \{o : o = \text{PERCEPT}(s) \text{ and } s \in b'\}$$

4.4.2 Searching with Observations

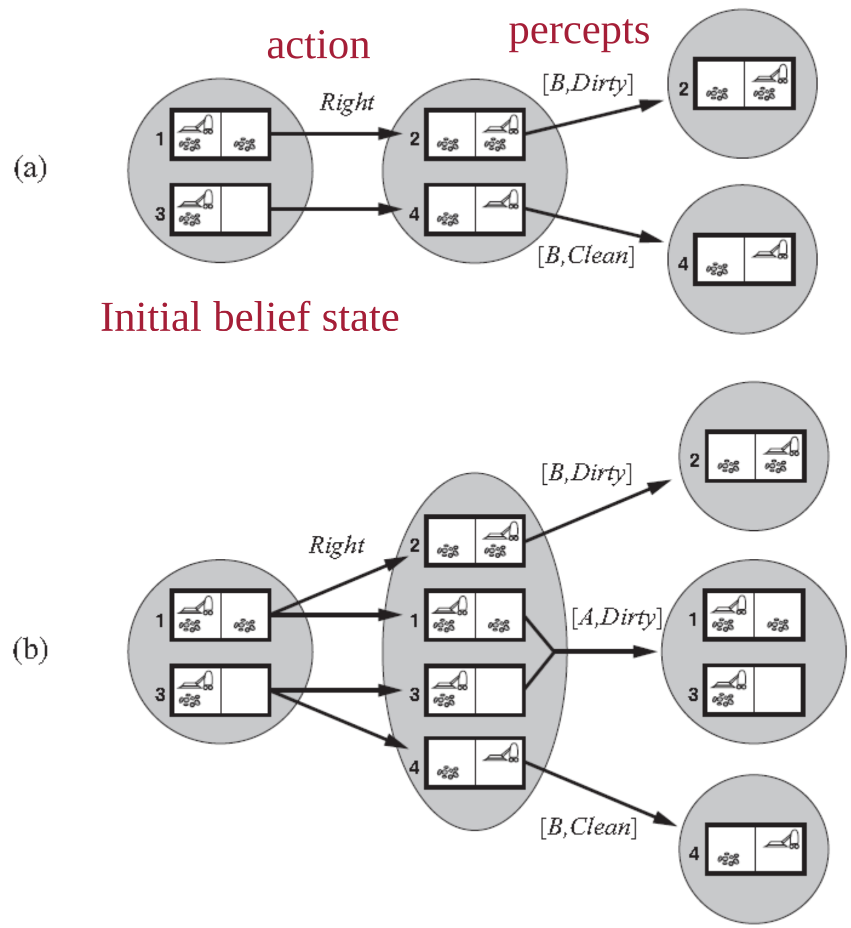
- Update: determines for each possible percept the belief state that would result from the percept. The new belief state b_o is the set of states in b' that could have produced the percept

$$b_o = \text{UPDATE}(b', o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in b'\}$$

- The updated belief state b_o can not be larger than the predicted belief b' state since observations can only help to reduce the uncertainty compared to the sensorless case
- For deterministic sensing, the belief states for the different percepts will be disjoint forming a partition of the original predicted belief state

4.4.2 Searching with Observations

- Transition in the deterministic case: *Right* is applied in the initial belief state resulting in a new belief state with two possible physical states depending on the percepts: [B,*Dirty*],[B,*Clean*]
- Transition in the slippery world: Applying *Right* gives a new belief state with four physical states leading to three belief states depending on the percepts: [B,*Dirty*], [A,*Dirty*], [B,*Clean*]



4.4.2 Searching with Observations

- Putting the three stages together, we obtain the possible belief states resulting from a given action and subsequent possible percept by:

$$\mathrm{RESULTS}(b,a) = \begin{cases} b_o : & b_o = \mathrm{UPDATE}(\mathrm{PREDICT}(b,a),o) \text{ and} \\ & o \in \mathrm{POSSIBLE-PERCEPTS}(\mathrm{PREDICT}(b,a)) \end{cases}$$

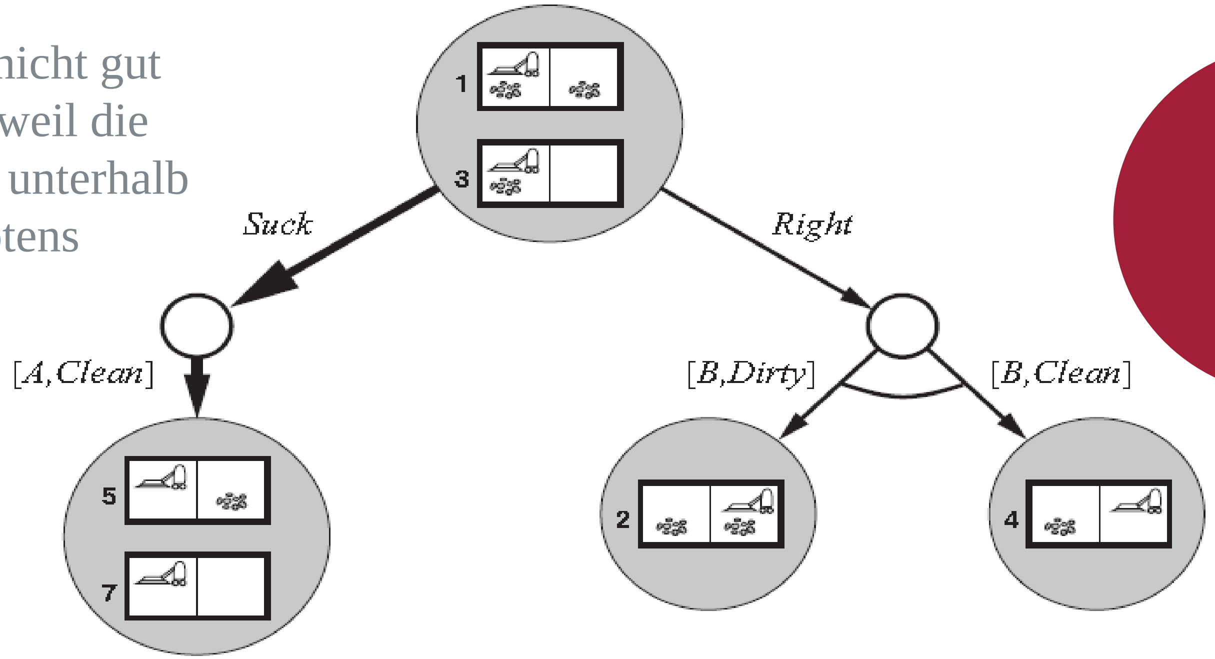
- Given this formulation AND-OR search can be applied directly to find a solution
- As in the sensorless case AND-OR search treats belief states as black boxes.
- The tree can be pruned by checking for subsets or supersets of the current state in the already generated belief states.

4.4.3 Solving partially observable problems

- First level of an AND-OR search tree for the initial percept [A,*Dirty*]
- The solution is the conditional plan

[*Suck*, *Right* **if** *BState*={6} **then** *Suck* **else** []]

(Lösung passt nicht gut zum Graphen, weil die if-Abfrage erst unterhalb des linken Knotens auftaucht.)

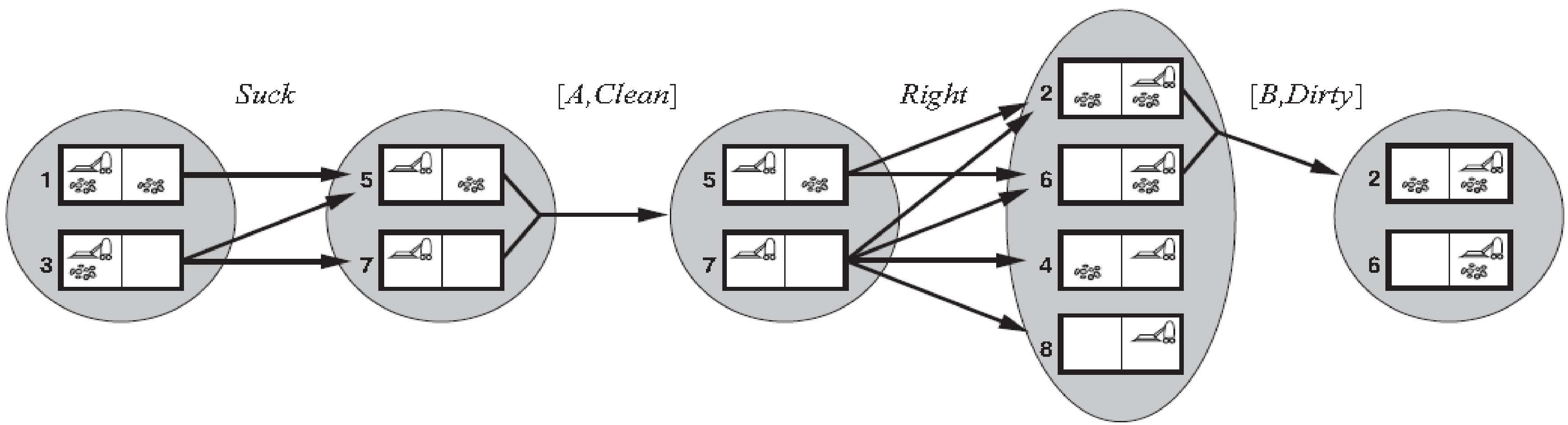


4.4.4 Partially observable environments

- An agent for partially observable problems is similar to the simple problem solving agent using the following elements
- The main differences are:
  - The solution is a conditional plan instead of a sequence
  - The agent has to maintain its belief state as it performs actions and receives percepts
- Maintaining the belief state is similar to the update step in the search phase and can be given a belief state *b*, an action *a*, and an observation *o* written as:

$$b' = \mathrm{UPDATE}(\mathrm{PREDICT}(b, a), o)$$

- Consider a slightly modified vacuum world where a square can become dirty again unless the agent is cleaning it actively at the time (kindergarten world).
- The resulting belief states for two update-prediction cycles for an agent in the kindergarten vacuum world are

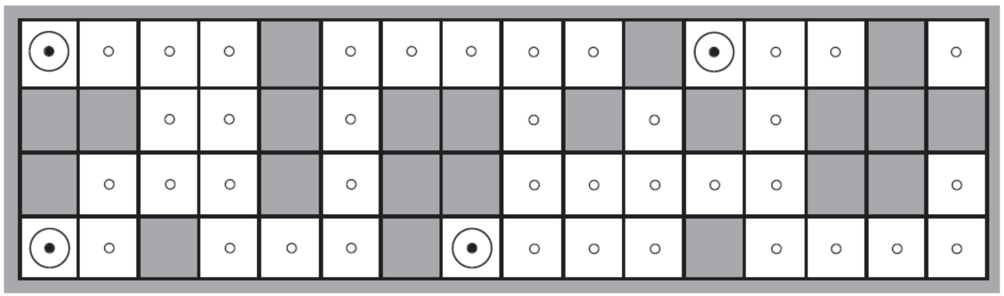


4.4.4 Partially observable environments

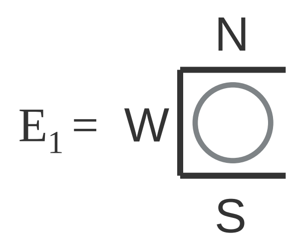
- Maintaining the belief state is a core function for any intelligent system in a partially observable environment – which includes the vast majority of real world problems
- This function is also called: monitoring, filtering, or state estimation
- The update-prediction step has to be as fast as the percepts are coming in, otherwise the agent falls behind
- In more complex environments an exact computation becomes unfeasible and approximations have to be used

4.4.4 Partially observable environments

- Localization is a problem with partial observations
- E.g. consider determining the possible positions of an erratic robot in a labyrinth with 4 sensors indicating the state of the cells in its 4-neighborhood

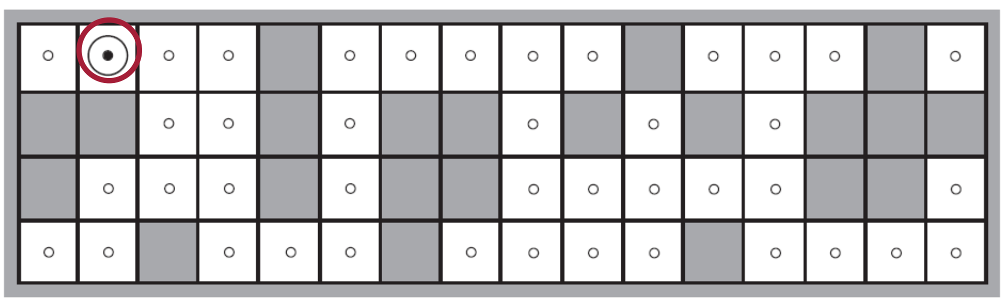


(a) Possible locations of robot after $E_1 = NSW$

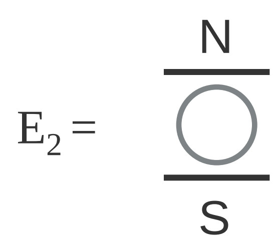


Move
(nondeterministic)

Robot must be here, after E_1, Move, E_2.



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



4.5 Online Search Agents

- So far we have examined offline search algorithms which compute a complete solution before acting
- In contrast we now consider online search agents which interleave computation and action
- Online search is a good idea in dynamic and semidynamic domains
- It is also helpful in nondeterministic domains because it allows the computational effort on the contingencies that actually arise rather than those that might happen
- Online search is necessary for unknown environ-ments where the agent does not know what states exist or what an action does (exploration problems)

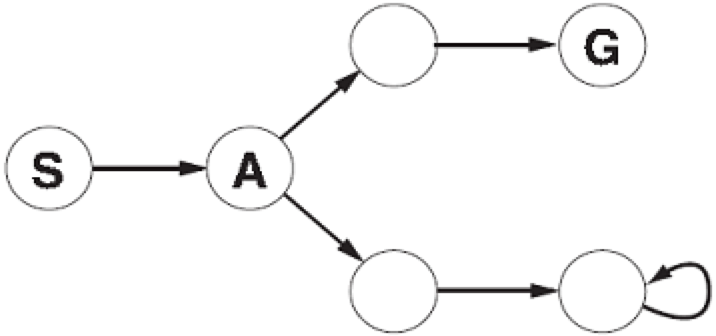
4.5 Online Search Problems

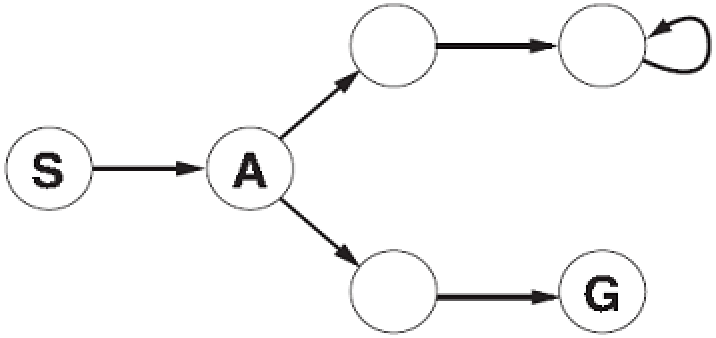
- A typical example for exploration problems is a robot that is placed in a new building where it has to build a map that it can use to get from *A* to *B*
- First we assume a deterministic and fully observable environment with
  - ACTIONS(*s*): returns a list of all allowed actions in state *s*
  - Step-cost function $c(s,a,s')$
  - GOAL-TEST(*s*)
- NOTE: We can not determine RESULT(*s*,*a*) except by actually doing action *a* in state *s*

4.5 Online Search Problems

- We may also have an admissible heuristic *h*(*s*) estimating the cost to goal G
- The objective is typically to reach a goal state with minimal cost
- A common performance measure is the competitive ratio which compares the actual cost with the best cost that would be achievable if the agent knows the whole state space in advance
- The best achievable cost ratio can be infinite if some actions are irreversible and the search reaches a dead-end state from which no goal is reachable

- No algorithm can avoid dead-ends in all state spaces
- For an agent both dead-end spaces look identical if it has visited only S and A so it has to take the same decision for both spaces: it will fail in one of them





4.5 Adversary Argument

- We now assume a safely explorable state space
- This means that a goal state is reachable from any reachable state
- State spaces with reversible actions like 8-queens or mazes can be viewed as undirected graphs and are safely explorable
- Even for safely explorable environments no bounded competitive ratio can be guaranteed if there are paths of unbounded cost

4.5 Online DFS Agent

- After each action the agent receives a percept telling it the reached state
- This can be used to augment the agent's map of the environment
- The map can be used to decide the next action
- This interleaving is different from offline agents which can immediately expand the next node
- An online agent can only expand the node it physically occupies
- To avoid traveling all over the search tree to expand the next node it is better to expand the nodes in local order

4.5 Online DFS Agent

- Depth-first-search works locally since the next node is always a child of the current node

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

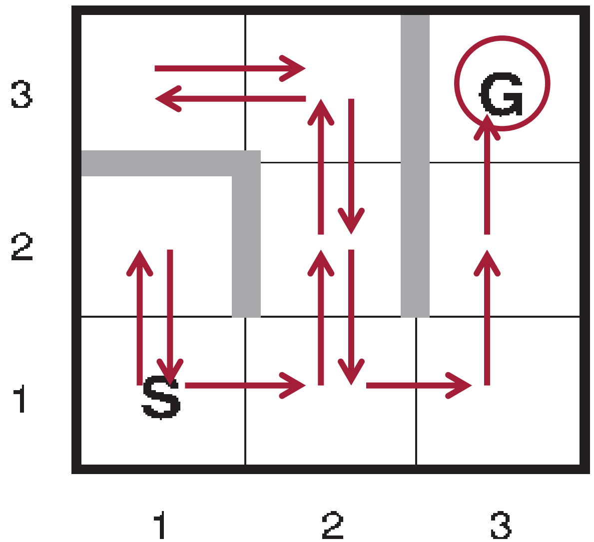
  if GOAL-TEST(s') **then return** $stop$
  if s' is a new state (not in $untried$) **then** $untried[s'] \leftarrow$ ACTIONS(s')
  if s is not null **then**
    $result[s, a] \leftarrow s'$
    add s to the front of $unbacktracked[s']$
  if $untried[s']$ is empty **then**
    if $unbacktracked[s']$ is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

4.5 Online DFS Agent

- The agent stores its map in a table RESULT[*s*,*a*] that records the state resulting from executing action *a* in state *s*
- When the agent reaches an action that has not been tried in the state it executes this action
- When all actions of the current state have been tried the agent has to physically backtrack instead of just dropping the node
- Backtracking means going back to the state from which the agent most recently entered the current state

4.5 Online DFS Agent

- When put in a maze the agent will in the worst case traverse every link twice
- For exploration this is optimal
- For finding a goal the competitive ratio can be arbitrarily bad if the goal is right next to the initial state
- An online variant of iterative deepening solves this problem
- ONLINE-DFS-AGENT only works in state spaces with reversible actions

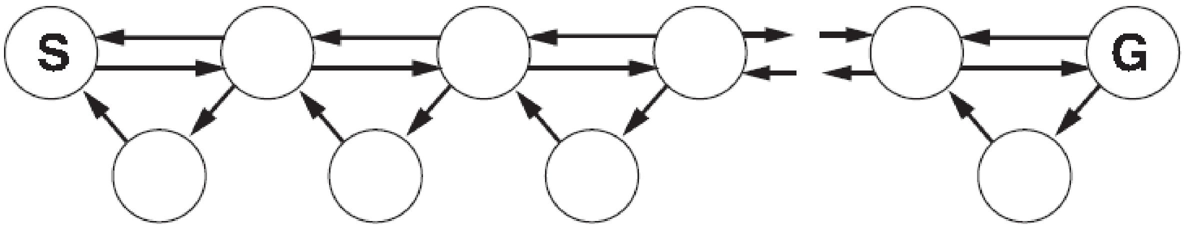


4.5 Online Local Search

- Hill-climbing search is already an online search algorithm because:
  - It only keeps the current state in memory, which can be identical to the physical state of the agent
  - It only searches locally, therefore it only visits reachable neighbor states
- In its simplest form it is not very useful since it may leave the agent at a local maximum
- Random restarts can not be used, because the agent can not teleport to a random new state

4.5 Random Walk

- Instead of random restarts random walk can be used to explore the environment
- It can be proven that random walk will eventually find a goal if the state space is finite
- The search can be extremely inefficient
- Consider a state space where backward steps are twice as likely as forward progress
- Random walk will take exponentially many steps to find the goal

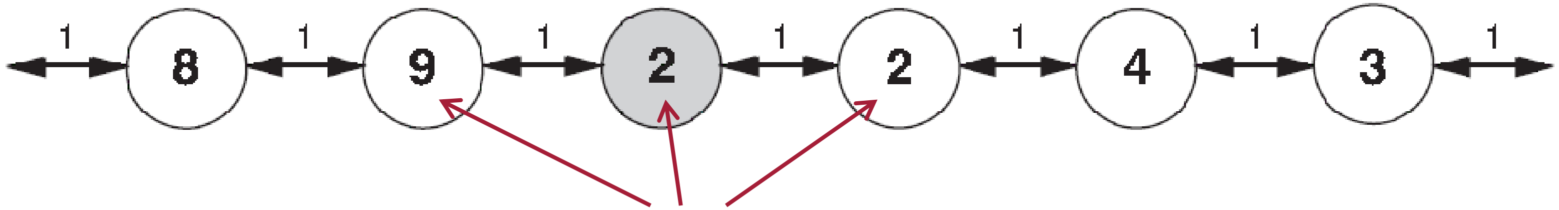


4.5 Random Walk

- Augmenting random walk with memory can be very effective
- The basic idea is to store a current best estimate of the cost to reach the goal from each state s that has been visited: *H(s)*
- Without information *H(s)* starts as the heuristic *h(s)*
- *H(s)* is updated as the agent gains experience in the state space

4.5 Learning Real-Time A*

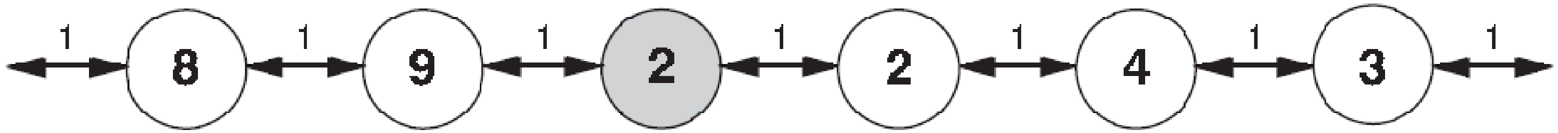
- Consider a simple example in a 1D state space



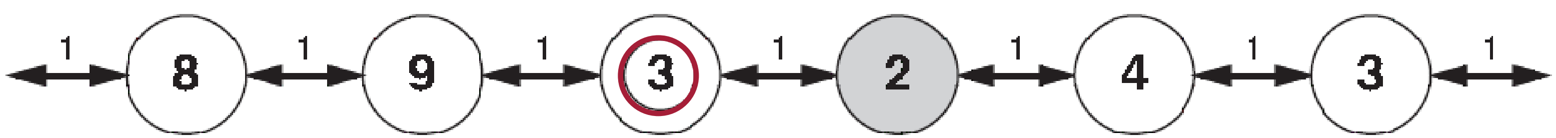
- The estimated cost *H(s)* is denoted in each state
- The agent seems to be stuck in a local minimum (shaded area)
- The estimated cost to reach the goal through a neighbor *s’* is the cost *c(s,a,s’)* to get to *s’* plus the estimated cost *H(s’)* to get to the goal from *s’*

$$\text{cost\_estimate}(s,a,G) = c(s,a,s') + H(s')$$

4.5 Learning Real-Time A*

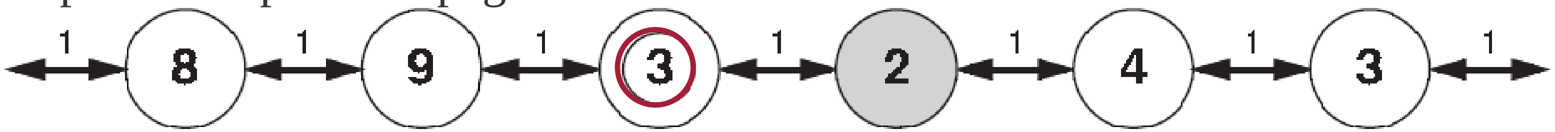


- For this state, there are two possible actions
  - Left with the estimated cost *1+9*
  - Right with the estimated cost *1+2*
- The best move seems to move to the right
- *H(s)=2* was overly optimistic and has to be updated to *H(s)=3*

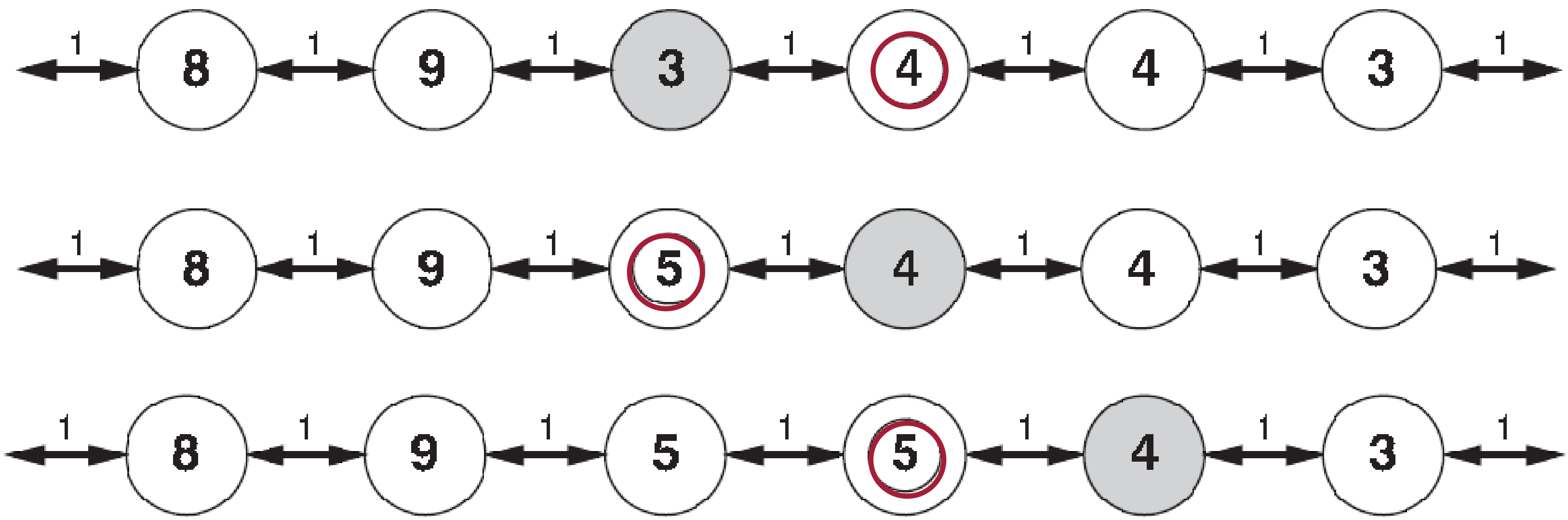


4.5 Learning Real-Time A*

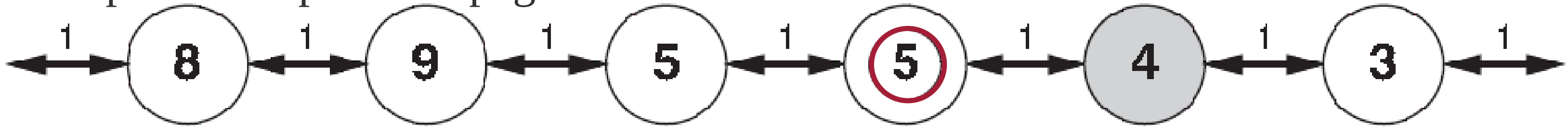
copied from previous page



- Continuing this, the agent moves back and forth twice more updating *H* each time



copied from previous page



- The agent can now leave the local minimum to the right
- An agent implementing this scheme is called learning real-time A* (LRTA*)
- Like ONLINE-DFS-AGENT it builds a map of the environment in the result table
- It updates the cost estimate of the state that it has just left and then chooses the apparently best move as next move

4.5 Learning Real-Time A*

- Actions that have not been tried are always assumed to lead to the goal with minimal cost *h(s)*
- This optimism under uncertainty encourages the agent to explore new possible actions
- An LRTA* agent is guaranteed to find a goal in any finite safely explorable environment
- Unlike A* it is not complete for infinite state spaces: there are cases where it can be led infinitely astray
- It can explore an environment with n states in $O(n^2)$

4.5 Learning Real-Time A*

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** $H[s'] \leftarrow h(s')$
  if s is not null
    $result[s, a] \leftarrow s'$
    $H[s] \leftarrow \min\limits_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(s, b, result[s, b], H)$
  $a \leftarrow$ an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, $result[s', b]$, H)
  $s \leftarrow s'$
  return a

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

4.5 Learning in Online Search

- The initial ignorance of online search provides several opportunities for learning
  - The agent can learn a “map” of the environment - the outcome of each action in each state
  - The agent can acquire more accurate estimates of the cost for each state by using local update rules
- Once exact values are known, optimal decisions can be taken by simply moving to the lowest cost successor; pure hill climbing is then the optimal strategy

Chapter 4 Summary 1

- Local search methods like hill climbing operate on complete state formulations, keeping only a small number of nodes in memory
- Several stochastic algorithms including simulated annealing have been developed
- Many local search methods also apply to continuous spaces
- Linear programming and convex optimization problems obey certain restrictions and allow polynomial-time algorithms

Chapter 4 Summary 2

- A genetic algorithm is a stochastic hill climbing search which holds a population of states and generates new states by mutation and crossover
- AND-OR search can generate contingency plans to reach the goal in nondeterministic environments
- Belief states represent the set of possible physical states in partially observable environments
- Standard search algorithms can be directly applied to belief states to solve sensorless problems
- Belief AND-OR search can solve partially observable problems

Chapter 4 Summary 3

- Incremental algorithms that construct solutions state by state within a belief state are often more efficient
- In exploration problems the agent has no knowledge about the states and actions
- Online search agents can build a map and find a goal if one exists
- Updating heuristic estimates provides an efficient method to escape from local minima