Artificial Intelligence

Local Search

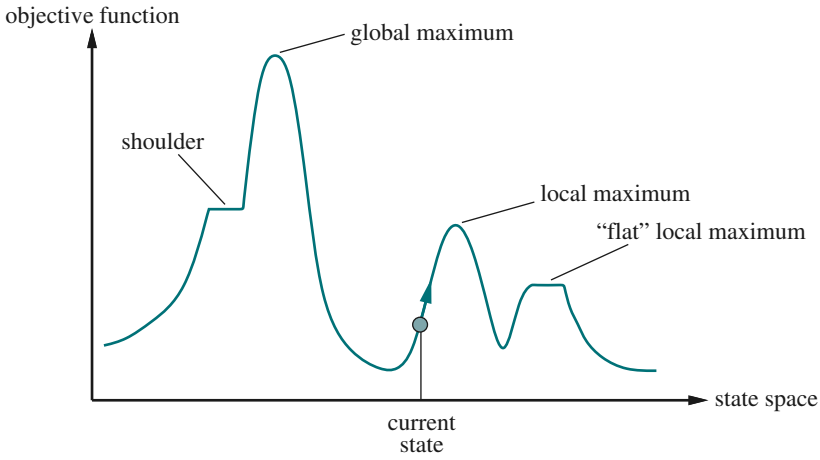
Christopher Simpkins

Local Search

If you don't care about the path to a goal state, you can use **local search**.

- Search neighbors of current state, moving to best neighbor.
- Track only current state.
- Uses very little memory.
- Can find reasonable solutions in large or infinite state spaces.
- Often used for **optmization** problems – finding states that maximize or minimize an **objective function**.

State Space Landscape



Hill-Climbing Search

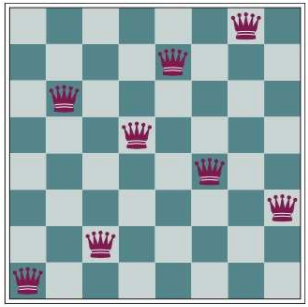
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current*
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    current ← *neighbor*

- Also known as **greedy local search**

The 8 Queens Problem

Complete-state formulation: row position for each of 8 columns, e.g., (a) below is

```
<1, 6, 2, 5, 7, 4, 8, 3>
```



(a)

(b)

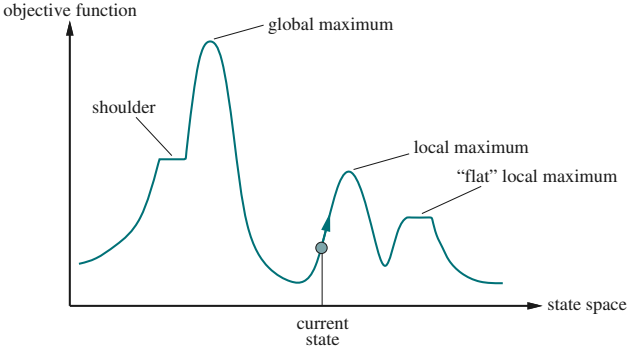
- Action: move a single queen to new row within column. Each state has $8 \cdot 7 = 56$ successor states.
- Possible heuristic: number of pairs of attacking queens (even if blocked). (b) above has $h = 17$.
  - Useful to remember: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

KENNESAW STATE UNIVERSITY

Disadvantages of Hill-Climbing

Susceptible to getting stuck in:

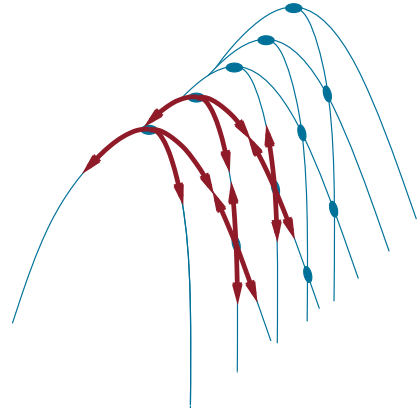
- local maxima
- ridges – sequences of local maxima
- plateaus, e.g., flat local maxima or shoulders.



How to fix:

- Allow "sideways" moves
- Stochastic hill climbing chooses randomly from uphill moves.
- Random restart hill climbing restarts from multiple initial states.

Grid of states superimposed on ridge rising from left to right.



Simulated Annealing

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 $e^{\Delta E/T}$

- Based on annealing in metallurgy – gradually cooling metals or glass to reach a low-energy crystalline state.
- Think in terms of gradient descent.
- Similar to hill climbing, but picks a random move and
  - accepts it if its better,
  - if not better, accept with probability < 1.
- Probability of accepting a worse move depends on:
  - how much worse the move is, ΔE, and
  - the current "temperature," T.

If T decreases sufficiently slowly, then the Boltzman distribution, $e^{\frac{\Delta E}{T}}$, ensures that all the probability is concentrated on the global maxima, so the algorithm finds a global maximum with probability approaching 1.

Evolutionary (Genetic) Algorithms

A kind of **local beam search**: tracking k states instead of just one.

Elements of genetic algorithms:

- Fitness function.
- Population size.
- Candidate representation:
  - Typically a string (vector) over a finite alphabet.
  - **Evolution strategies**: sequence of real numbers.
  - **Genetic programming**: computer programs.
- Mixing number, ρ: number of "parents" from which to generate new candidates. When $\rho = 1$, stochastic beam search.
- Selection process for choosing "parents."
- Recombination procedure.
- Mutation rate.
- Composition of next generation.
  - Elitism: choose top-scoring candidates.
  - Culling: eliminate bottom-scoring candidates.

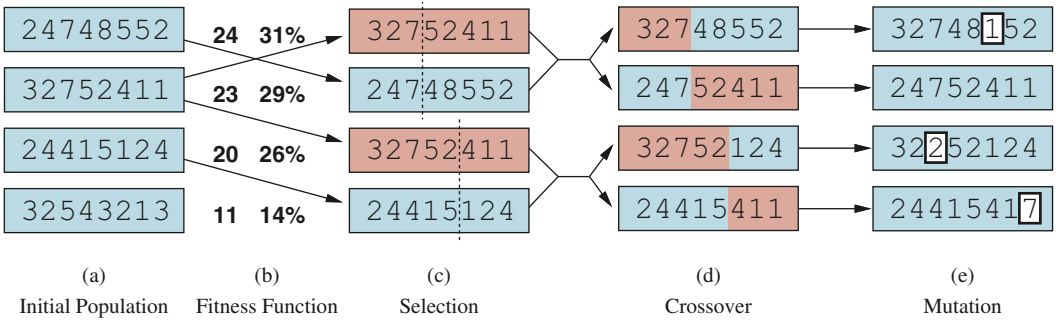
KENNESAW STATE UNIVERSITY

A Genetic Algorithm

function GENETIC-ALGORITHM(*population*, *fitness*) **returns** an individual
  repeat
    weights ← WEIGHTED-BY(*population*, *fitness*)
    population2 ← empty list
    for $i = 1$ **to** SIZE(*population*) **do**
      parent1, *parent2* ← WEIGHTED-RANDOM-CHOICES(*population*, *weights*, 2)
      child ← REPRODUCE(*parent1*, *parent2*)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *population2*
    population ← *population2*
  until some individual is fit enough, or enough time has elapsed
  return the best individual in *population*, according to *fitness*

function REPRODUCE(*parent1*, *parent2*) **returns** an individual
  n ← LENGTH(*parent1*)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(*parent1*, 1, c), SUBSTRING(*parent2*, $c+1$, n))

Genetic Algorithm on 8-Queens Problem

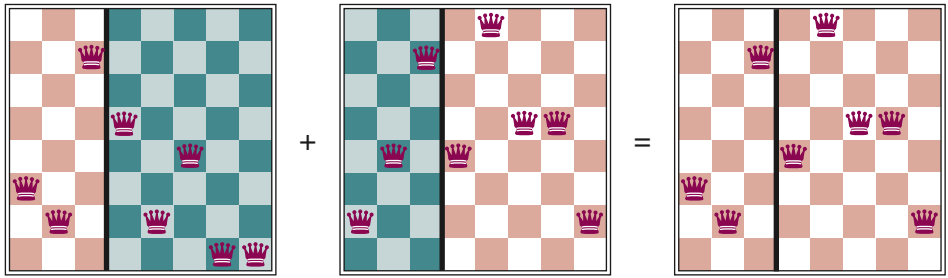


1. Population is generated in (a).
2. Fitness function is applied to population, which is then ranked by fitness score.
3. Highest-scoring candidates are selected for reproduction in (c).
4. Crossover operation is applied to candidates in (c) to produce "children" in (d).
5. In (e) "offspring" are randomly chosen for mutation. For each chosen candidate, a "gene" is randomly chosen, then that gene is assigned a random "mutated" value.

KENNESAW STATE UNIVERSITY

Crossover in the 8-queens Problem

Here is a pictorial illustration of the crossover operation in the 8-queens problem:



Is the random crossover operation depicted here meaningful for the 8-queens problem?

Genetic Algorithms and Biological Evolution

Genetic algorithms borrow the language of biological evolution for marketing purposes, but are far more simplistic than biological evolution. My takes:

- Genetic algorithms are just stochastic beam search with "sexual" successor generation and "mutation."
- If there is no meaningful crossover operation, genetic algorithms are just random walks in the state space graph.

There is an interesting connection between biological evolution and AI, in particular learning.

- Learning is adaptation. With experience an agent adapts to a task, getting better at the task.
- Biological evolution can be seen as a learning process whereby specieses "learn" to perform better in their environments.
- The Baldwin effect: immutable traits vs. online learning ability.
  - Plasticity, or the ability to learn, allows a species to adapt to an environment for which it is ill-suited. E.g., building shelters, fire, etc. in cold regions.
  - Things that are harder, or impossible, to learn online must be encoded in the genome. E.g., the way our body uses the air it breathes or the sun.

Continuous State Spaces – Airports In Romania

If each airport $\boldsymbol{x}_i$ is at location (x_i, y_i) and the set of cities closest to airport $\boldsymbol{x}_i$ is C_i, then

$$f(\boldsymbol{x}) = f(x_1, y_1, x_2, y_2, x_3, y_3)$$

and we want to minimize

$$f(\boldsymbol{x}) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

For a globally optimal solution, if the airports move "too much," the sets C_i change. How to deal with that?

KENNESAW STATE UNIVERSITY

Local Gradient Descent

The gradient of

$$f(\boldsymbol{x}) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

is

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

But that would only work for one airport. We can decompose it into three local problems:

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_i} (x_1 - x_c) \qquad \frac{\partial f}{\partial x_2} = 2 \sum_{c \in C_i} (x_2 - x_c) \qquad \frac{\partial f}{\partial x_3} = 2 \sum_{c \in C_i} (x_3 - x_c)$$

$$\frac{\partial f}{\partial y_1} = 2 \sum_{c \in C_i} (y_1 - y_c) \qquad \frac{\partial f}{\partial y_2} = 2 \sum_{c \in C_i} (y_2 - y_c) \qquad \frac{\partial f}{\partial y_3} = 2 \sum_{c \in C_i} (y_3 - y_c)$$

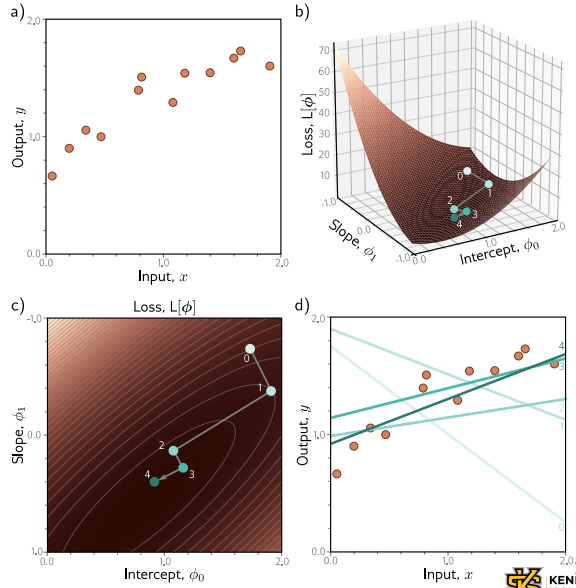
Gradient Descent in Action

Given our 3 gradient expressions, we can use the update rule:

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

where α is a **step size**, or learning rate.

- What if α is "too big?"
- What if α is "too small?"

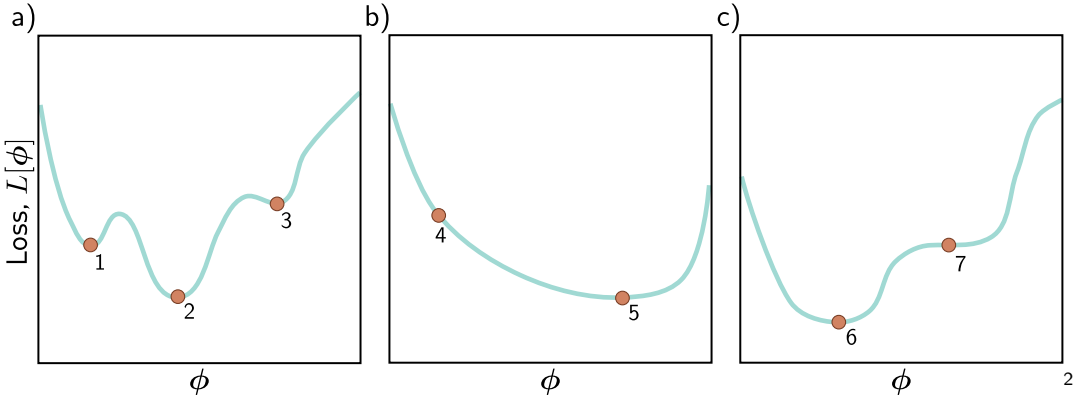


[1] https://udlbook.github.io/udlbook/

KENNESAW STATE UNIVERSITY

Continuous State Spaces and Convexity

A convex set is a set of points in which a line between any two points lies within the set. A convex function is a function for which the points above the function form a convex set.

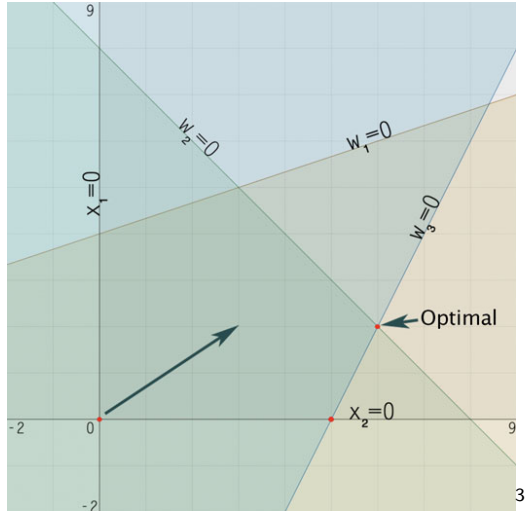


There are mathematical properties of continuous spaces that rule out local minima. Take my deep learning class to learn about them!

[2] https://udlbook.github.io/udlbook/

Constrained Optimization via Linear Programming

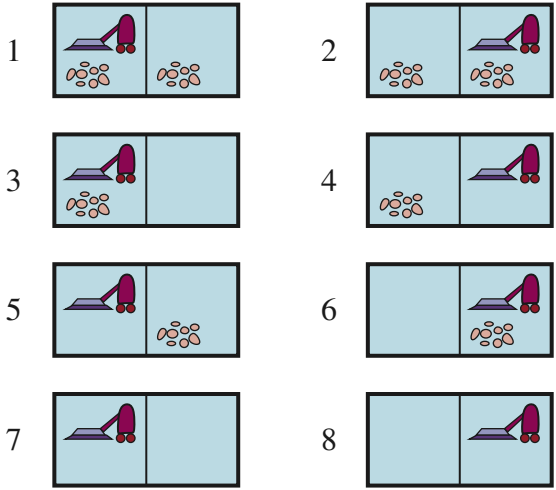
$$
\begin{aligned}
\text{maximize } & 3x_1 + 2x_2 \\
\text{subject to } & -x_1 + 3x_2 \leq 12 \\
& x_1 + x_2 \leq 8 \\
& 2x_1 - x_2 \leq 10 \\
& x_1, x_2 \geq 0
\end{aligned}
$$



[3]

[3] https://vanderbei.princeton.edu/LPbook/

States in the Vacuum World



Nondeterministic Actions: The Erratic Vacuum World

In the erratic vacuum world, the Suck action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.

So the result of each action is a set, e.g.:

$$\text{RESULTS}(1, Suck) = \{5, 7\}$$

A Factored Representation

Let's depart from the book and, instead of using an index into a vector of states, create a factored representation for clarity.

- `left-condition` $\in$ {`CLEAN`, `DIRTY`}$
- `right-condition` $\in$ {`CLEAN`, `DIRTY`}$
- `vacuum-location` $\in$ {`LEFT`, `RIGHT`}$
- State representation: `<vacuum-location, left-condition, right-condition>`

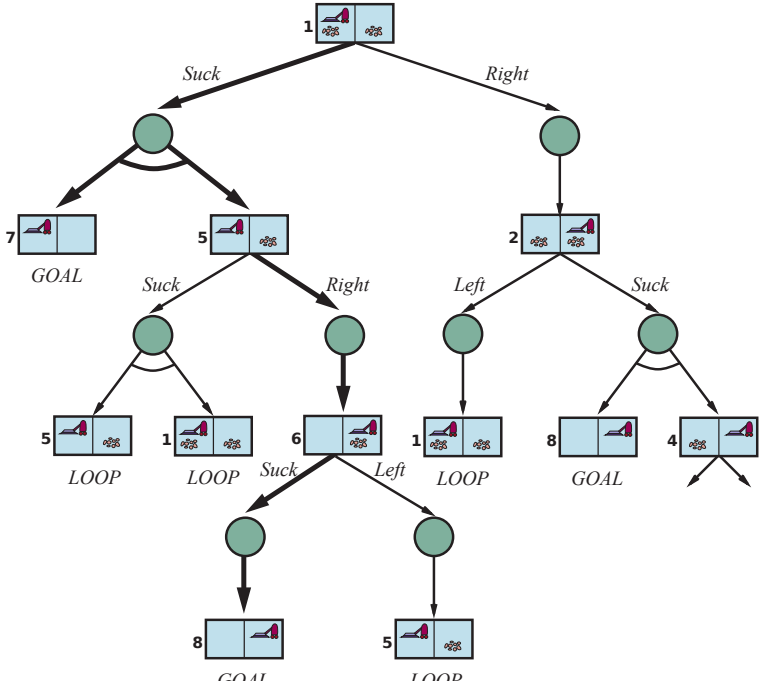
So

$$\text{RESULTS}(1, Suck) = \{5, 7\}$$

becomes

`RESULTS(<LEFT, DIRTY, DIRTY>, Suck)= {<LEFT, CLEAN, DIRTY>, <LEFT, CLEAN, CLEAN>}`

AND-OR Search Trees



AND-OR Search Algorithm

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

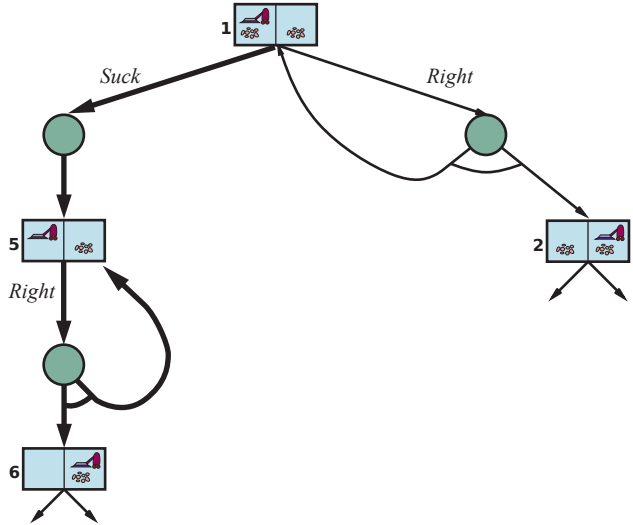
function OR-SEARCH(*problem*, *state*, *path*) **returns** *a conditional plan, or failure*
  if *problem*.IS-GOAL(*state*) **then return** the empty plan
  if IS-CYCLE(*state*, *path*) **then return** *failure*
  for each *action* **in** *problem*.ACTIONS(*state*) **do**
    plan $\leftarrow$ AND-SEARCH(*problem*, 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** $plan_n$]

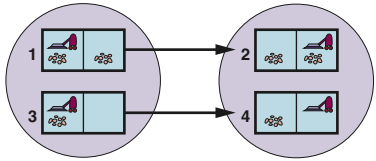
Slippery Vacuum World

Like deterministic vacuum world, but a movement action may result in no movement.

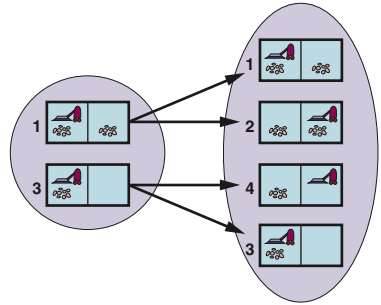
$$\text{RESULTS}(1, Right) = \{1, 2\}$$



Sensorless Vacuum Belief States

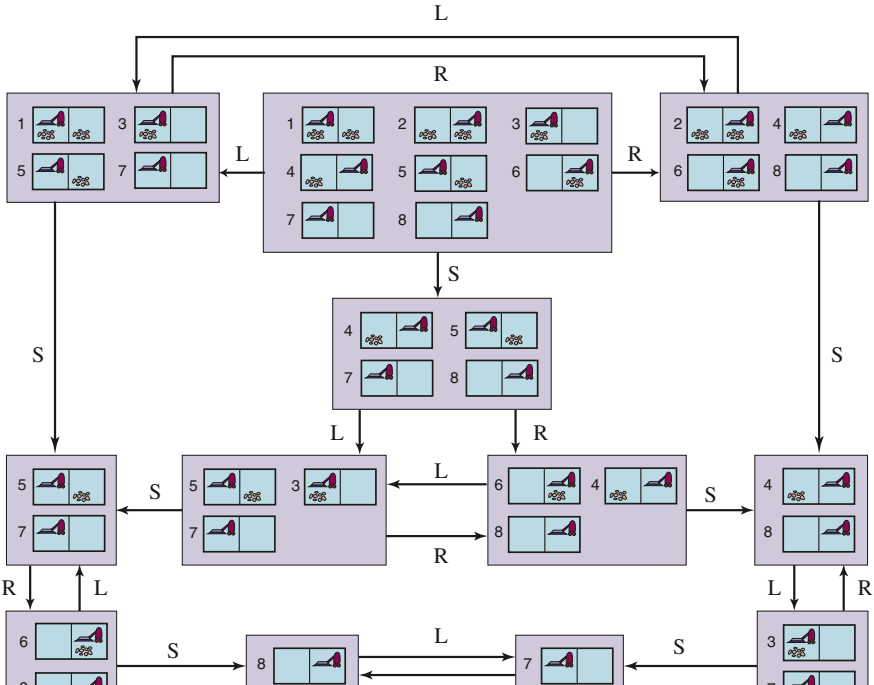


(a)



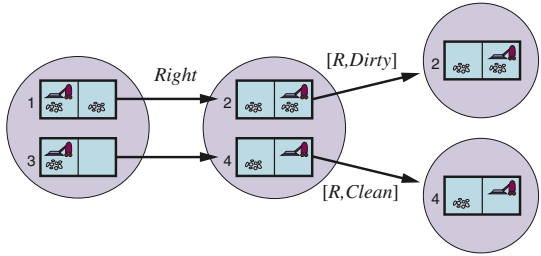
(b)

Reachable States in Sensorless Vacuum World

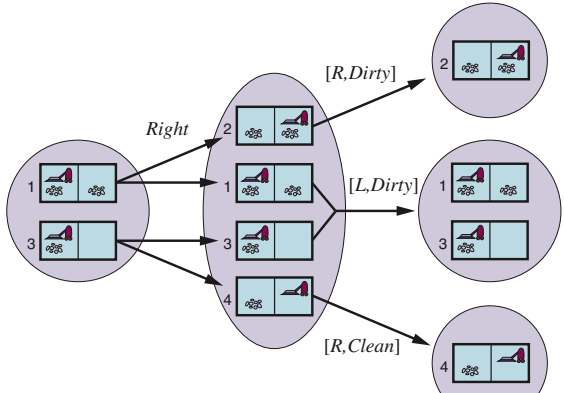


State Transitions with Local Sensing

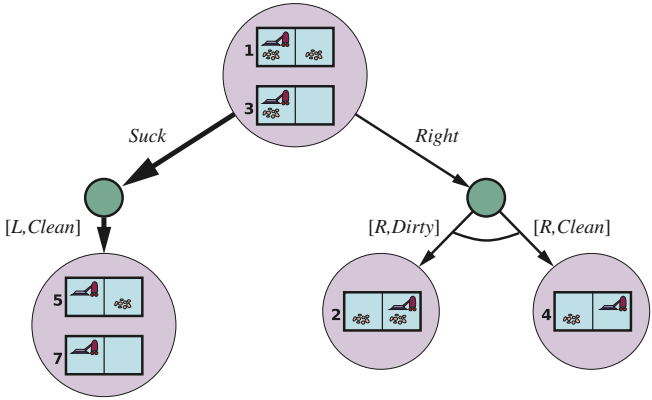
(a)



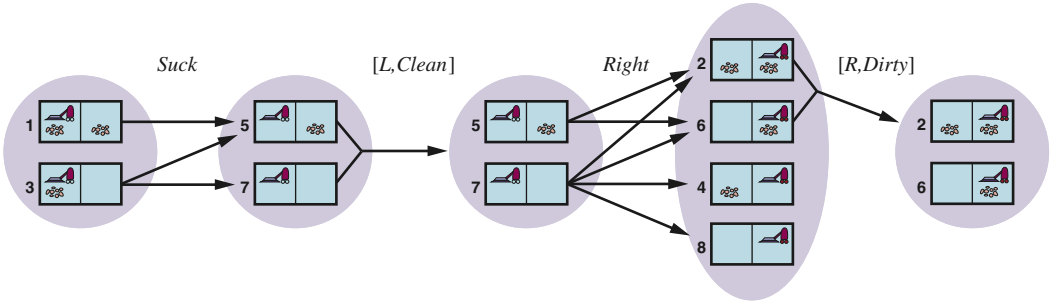
(b)



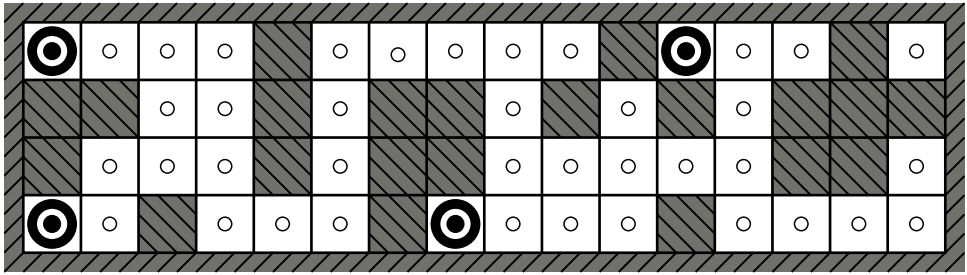
Local Sensing And-Or Trees



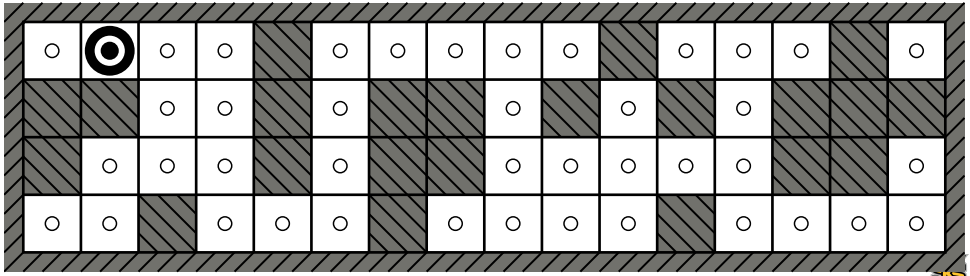
Prediction-Update Cycles



Robot Localization

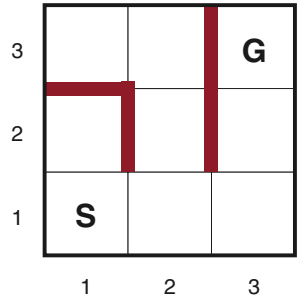


(a) Possible locations of robot after $E_1 = 1011$

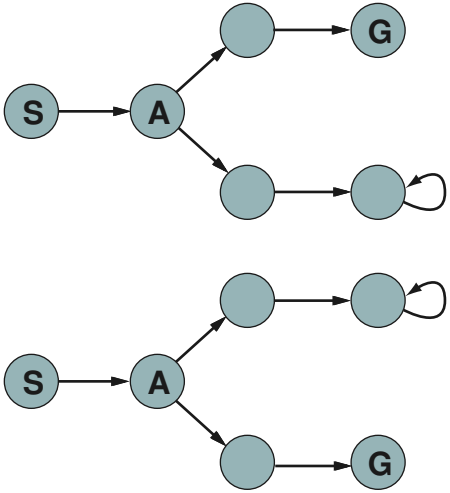


(b) Possible locations of robot after $E_1 = 1011, E_2 = 1010$

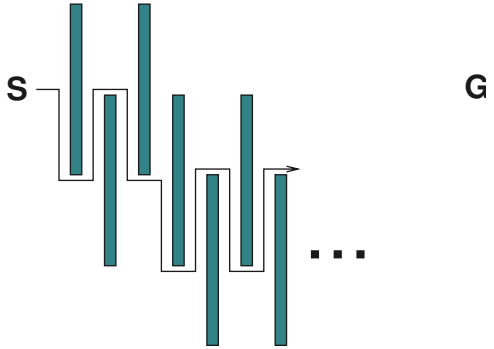
Maze Problems



Dead Ends in Online Search



(a)



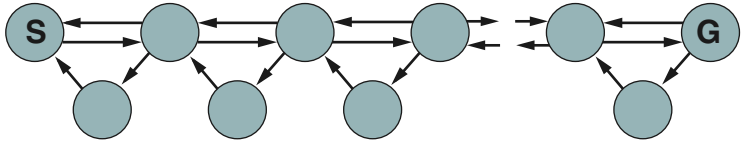
(b)

Online Depth-First Search

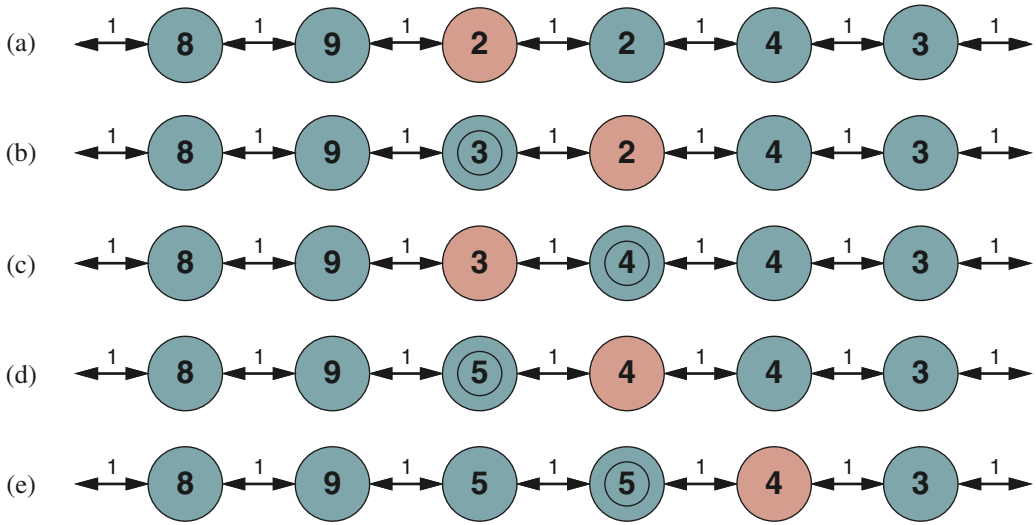
function ONLINE-DFS-AGENT(*problem*, s') **returns** an action
s, *a*, the previous state and action, initially null
result, a table mapping $(s,\ a)$ to s', initially empty
untried, a table mapping s to a list of untried actions
unbacktracked, a table mapping s to a list of states never backtracked to

if *problem*.IS-GOAL(s') **then return** *stop*
if s' is a new state (not in *untried*) **then** *untried*$[s'] \leftarrow$ *problem*.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*
    $a \leftarrow$ an action b such that *result*$[s', b]$ = POP(*unbacktracked*$[s']$)$s' \leftarrow$ null
else $a \leftarrow$ POP(*untried*$[s']$)
$s \leftarrow s'$
return a

Random Walks



$LRTA^*$ Iterations



$LRTA^*$ Algorithm

function LRTA*-AGENT(*problem*, s', h) **returns** an action
  s, a, the previous state and action, initially null
  result, a table mapping $(s,\ a)$ to s', initially empty
  H, a table mapping s to a cost estimate, initially empty

  if IS-GOAL(s') **then return** *stop*
  if s' is a new state (not in H) **then** $H[s'] \leftarrow h(s')$
  if s is not null **then**
    $result[s,a] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST(*problem*, $s, b, result[s,b], H$)
  $a \leftarrow \text{argmin}_{b \in \text{ACTIONS}(s)}$ LRTA*-COST(*problem*, $s', b, result[s',b], H$)
  $s \leftarrow s'$
  return a

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