Beyond Classical Search

Juhan Ernits
Department of Computer Science
Tallinn University of Technology
Juhan.ernits@ttu.ee
2016

Local search algorithms and optimization

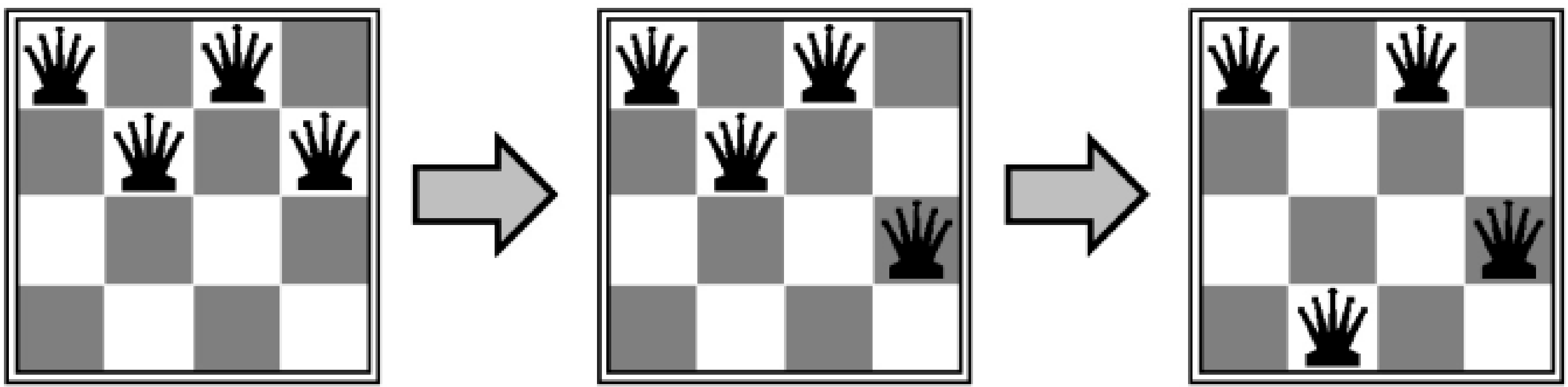
- Systematic search algorithms
  - to find (or given) the goal and to find the path to that goal

- Local search algorithms
  - the path to the goal is irrelevant, e.g., *n*-queens problem
  - state space = set of “complete” configurations
  - keep a single “current” state and try to improve it, e.g., move to its neighbors
  - Key advantages:
    - use very little (constant) memory
    - find reasonable solutions in large or infinite (continuous) state spaces
  - (pure) Optimization problem:
    - to find the best state (optimal configuration) based on an objective function, e.g. reproductive fitness – Darwinian, no goal test and path cost

Local search – example

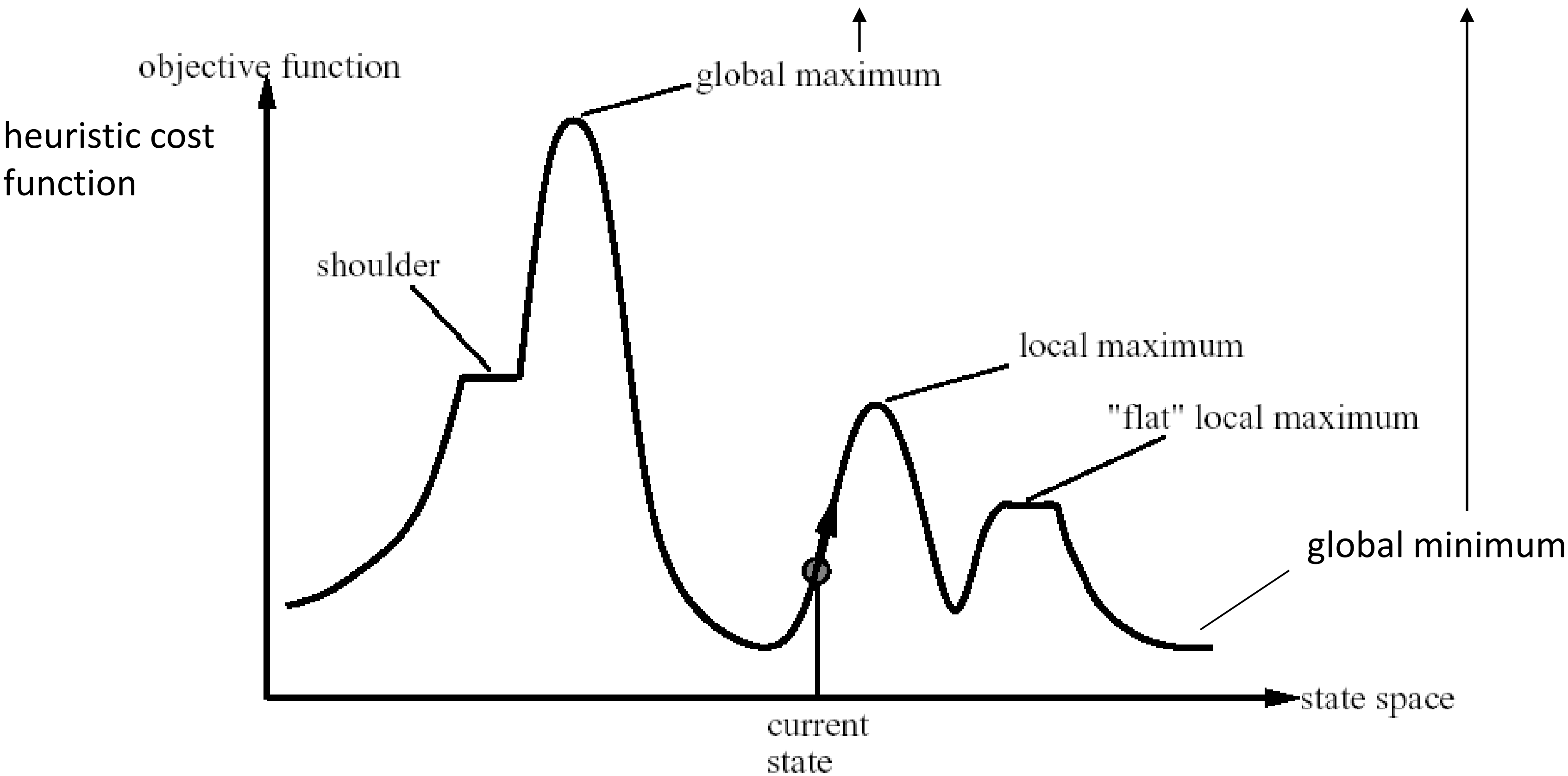
Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



Local search – state space landscape

- elevation = the value of the objective function or heuristic cost function



- A complete local search algorithm finds a solution if one exists
- A optimal algorithm finds a global minimum or maximum

Hill-climbing search

- moves in the direction of increasing value until a "peak"
  - current node data structure only records the state and its objective function
  - neither remember the history nor look beyond the immediate neighbors
  - like climbing Mount Everest in thick fog with amnesia

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] < VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

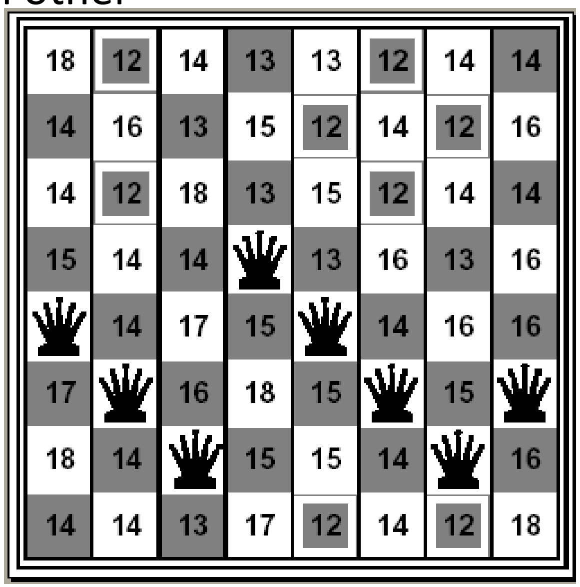
Hill-climbing search

- moves in the direction of increasing value until a "peak"
  - current node data structure only records the state and its objective function
  - neither remember the history nor look beyond the immediate neighbors
  - like climbing Mount Everest in thick fog with amnesia

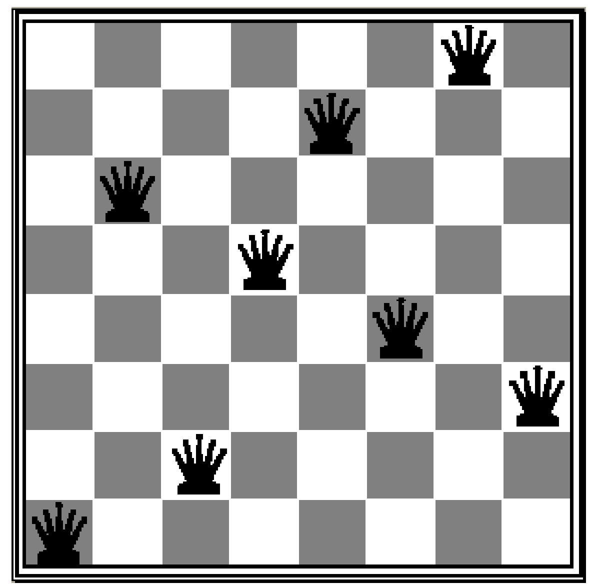
```
def hill_climbing(problem):
    """From the initial node, keep choosing the neighbor with highest value,
    stopping when no neighbor is better. [Fig. 4.2]"""
    current = Node(problem.initial)
    while True:
        neighbors = current.expand(problem)
        if not neighbors:
            break
        neighbor = argmax_random_tie(neighbors,
                                     lambda node: problem.value(node.state))
        if problem.value(neighbor.state) <= problem.value(current.state):
            break
        current = neighbor
    return current.state
```

Hill-climbing search - example

- complete-state formulation for 8-queens
  - successor function returns all possible states generated by moving a single queen to another square in the same column (8 x 7 = 56 successors for each state)
  - the heuristic cost function *h* is the number of pairs of queens that are attacking each other



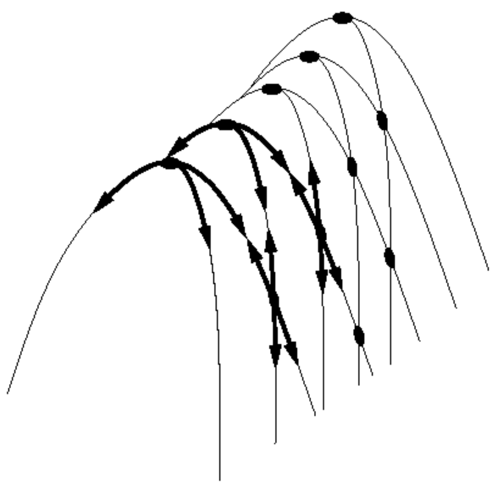
best moves reduce *h* = 17 to *h* = 12

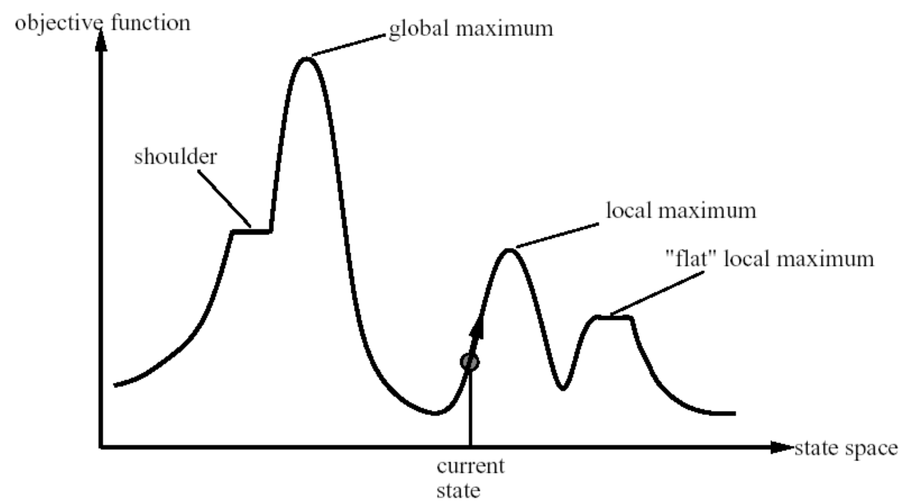


local minimum with *h* = 1

Hill-climbing search – greedy local search

- Hill climbing, the greedy local search, often gets stuck
  - Local maxima: a peak that is higher than each of its neighboring states, but lower than the global maximum
  - Ridges: a sequence of local maxima that is difficult to navigate





  - Plateau: a flat area of the state space landscape
    - a flat local maximum: no uphill exit exists
    - a shoulder: possible to make progress
  - can only solve 14% of 8-queen instance but fast (4 steps to *S* and 3 to *F*)

Hill-climbing search – improvement

- Allows sideways move: with hope that the plateau is a shoulder
  - could stuck in an infinite loop when it reaches a flat local maximum
  - limits the number of consecutive sideways moves
  - can solve 94% of 8-queen instances but slow (21 steps to *S* and 64 to *F*)
- Variations
  - stochastic hill climbing
    - chooses at random; probability of selection depends on the steepness
  - first choice hill climbing
    - randomly generates successors to find a better one
  - All the hill climbing algorithms discussed so far are incomplete
    - fail to find a goal when one exists because they get stuck on local maxima
  - Random-restart hill climbing
    - conducts a series of hill-climbing searches; randomly generated initial states
  - Have to give up the global optimality
    - landscape consists of a large amount of porcupines on a flat floor
    - NP-hard problems

Simulated annealing search

- combine hill climbing (efficiency) with random walk (completeness)
- annealing: harden metals by heating metals to a high temperature and gradually cooling them
- getting a ping-pong ball into the deepest crevice in a humpy surface
  - shake the surface to get the ball out of the local minima
  - not too hard to dislodge it from the global minimum
- simulated annealing:
  - start by shaking hard (at a high temperature) and then gradually reduce the intensity of the shaking (lower the temperature)
  - escape the local minima by allowing some “bad” moves
  - but gradually reduce their size and frequency

Simulated annealing search - Implementation

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps
   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(ΔE/T)
```

- Always accept the good moves $\Delta E > 0$
- The probability to accept a bad move
  - decreases exponentially with the "badness" of the move $\Delta E < 0$
  - decreases exponentially with the "temperature" *T* (decreasing)
- finds a global optimum with probability approaching 1 if the *schedule* lowers *T* slowly enough

Simulated annealing search - Implementation

```
def simulated_annealing(problem, schedule=exp_schedule()):
    "[Fig. 4.5]"
    current = Node(problem.initial)
    for t in xrange(sys.maxint):
        T = schedule(t)
        if T == 0:
            return current
        neighbors = current.expand(problem)
        if not neighbors:
            return current
        next = random.choice(neighbors)
        delta_e = problem.value(next.state) - problem.value(current.state)
        if delta_e > 0 or probability(math.exp(delta_e/T)):
            current = next
```

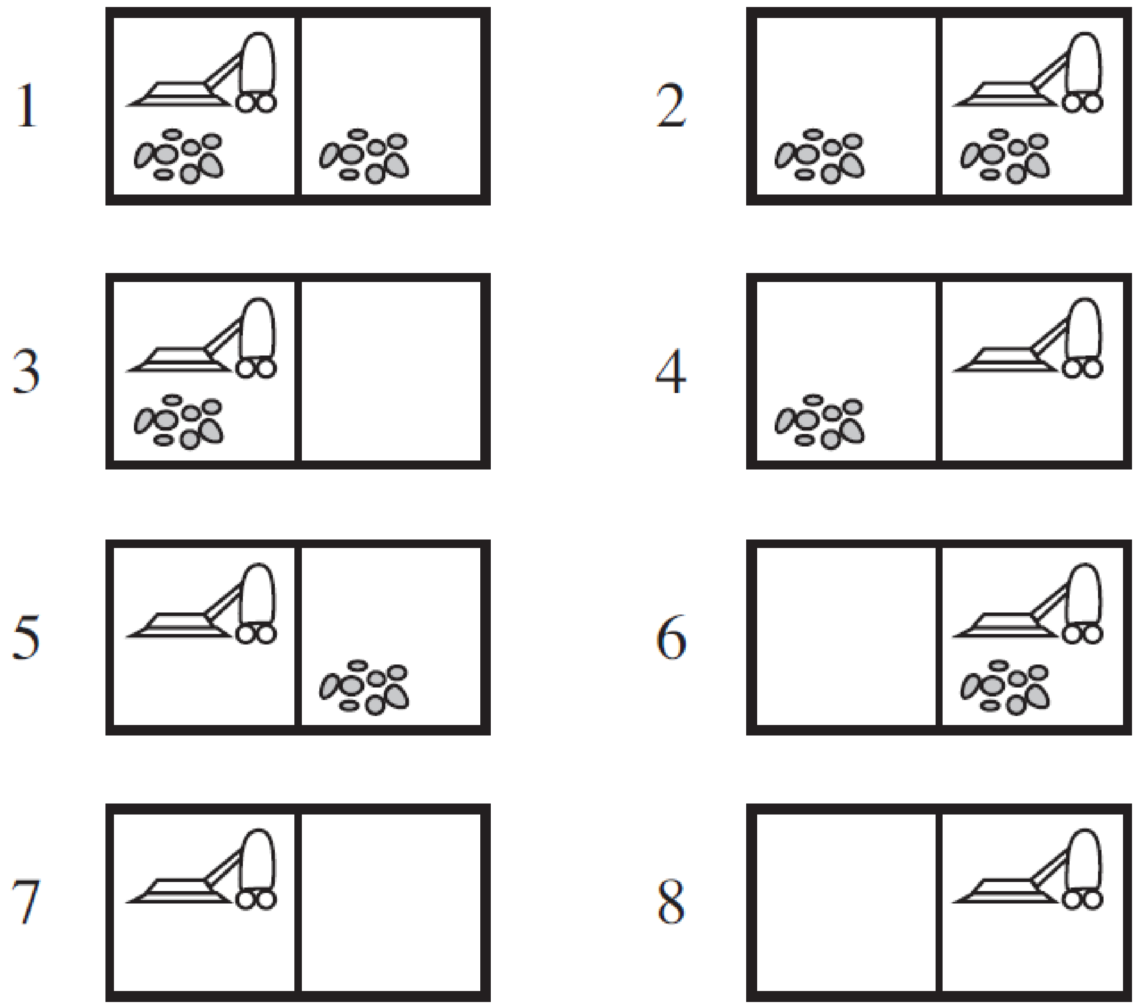
- Always accept the good moves $\Delta E > 0$
- The probability to accept a bad move
  - decreases exponentially with the "badness" of the move $\Delta E < 0$
  - decreases exponentially with the "temperature" *T* (decreasing)
- finds a global optimum with probability approaching 1 if the *schedule* lowers *T* slowly enough

Local beam search

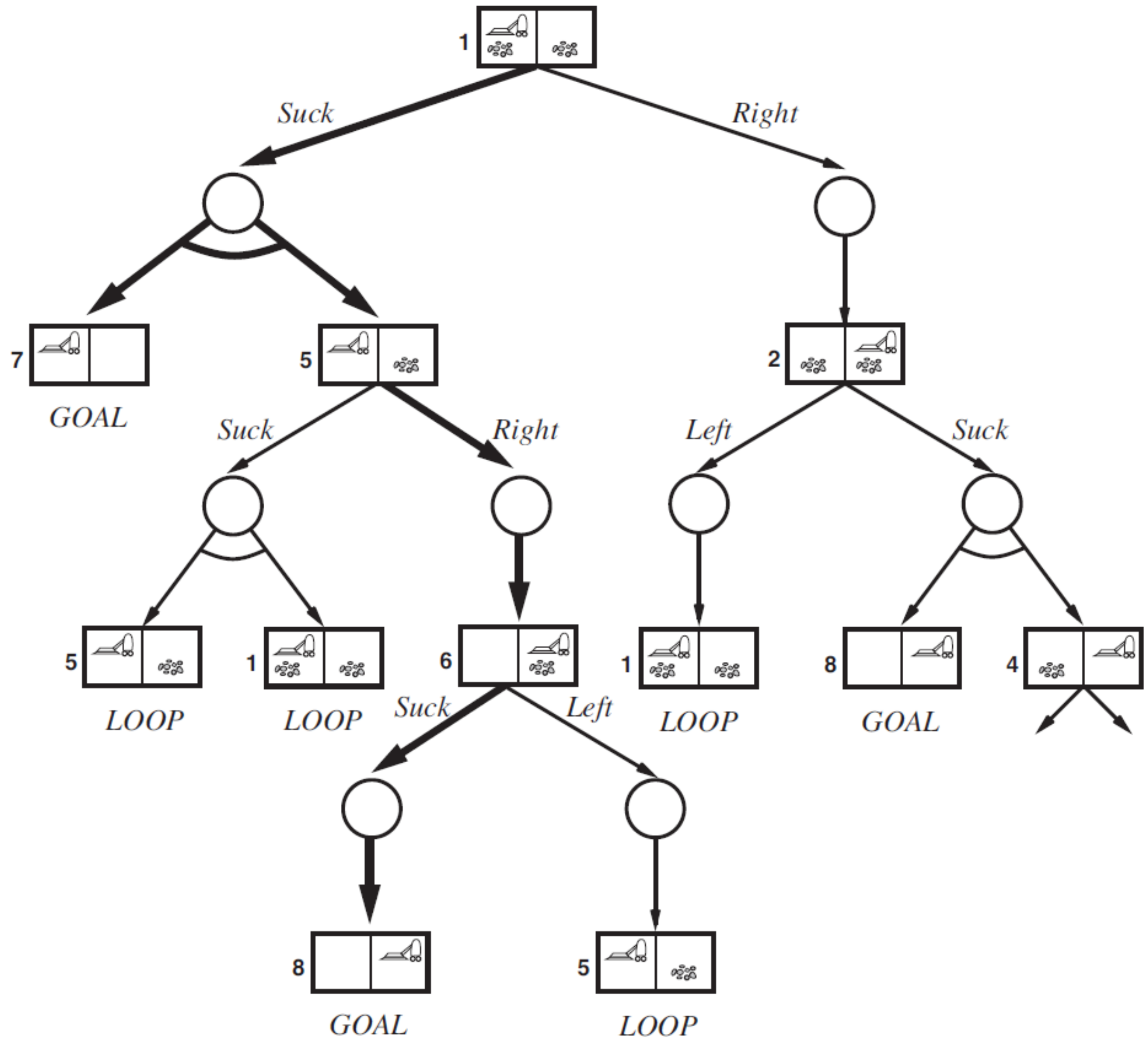
- Local beam search: keeps track of *k* states rather than just one
  - generates all the successors of all *k* states
  - selects the *k* best successors from the complete list and repeats
  - quickly abandons unfruitful searches and moves to the space where the most progress is being made
    – "Come over here, the grass is greener!"
  - lack of diversity among the *k* states
- stochastic beam search: chooses *k* successors at random, with the probability of choosing a given successor having an increasing value
- natural selection: the successors (offspring) if a state (organism) populate the next generation according to is value (fitness).

Search with nondeterministic actions

Sometimes the vacuum cleaner also cleans the neighbouring cell, sometimes releases dirt to a clean cell.



AND-OR trees



1
Suck
Right
7
GOAL
5
Suck
Right
2
Left
Suck
5
LOOP
1
LOOP
6
Suck
Left
1
LOOP
8
GOAL
4
8
GOAL
5
LOOP

Depth first AND-OR tree 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]
```

Slippery Vacuum World