Artificial Intelligence 1

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Problem Solving: Local Search, AND-OR Search, and On-line Search

We know how to use **heuristics in search**

- BFS, A*, IDA*, RBFS, SMA*
- looking for a sequence of actions in fully observable, deterministic, static, known environments

Next:

- **What if the path is not important?**
  - Local search: HC, SA, BS, GA
- **What if actions are non-deterministic?**
  - AND-OR search
- **What if the knowledge of world changes?**
  - on-line search, LRTA*

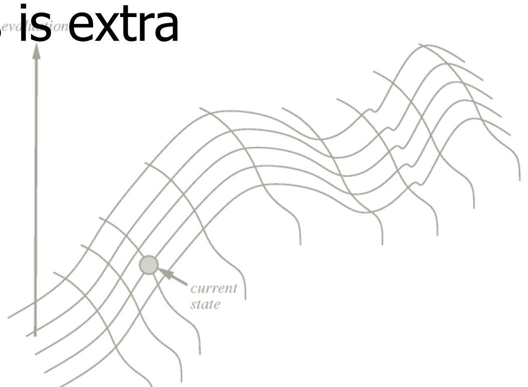
Local search

So far we systematically explored all paths possibly going to the goal and the path itself was a part of the solution.

For some problems (e.g. 8-queens) the path is not relevant to the problem, only the goal is important.

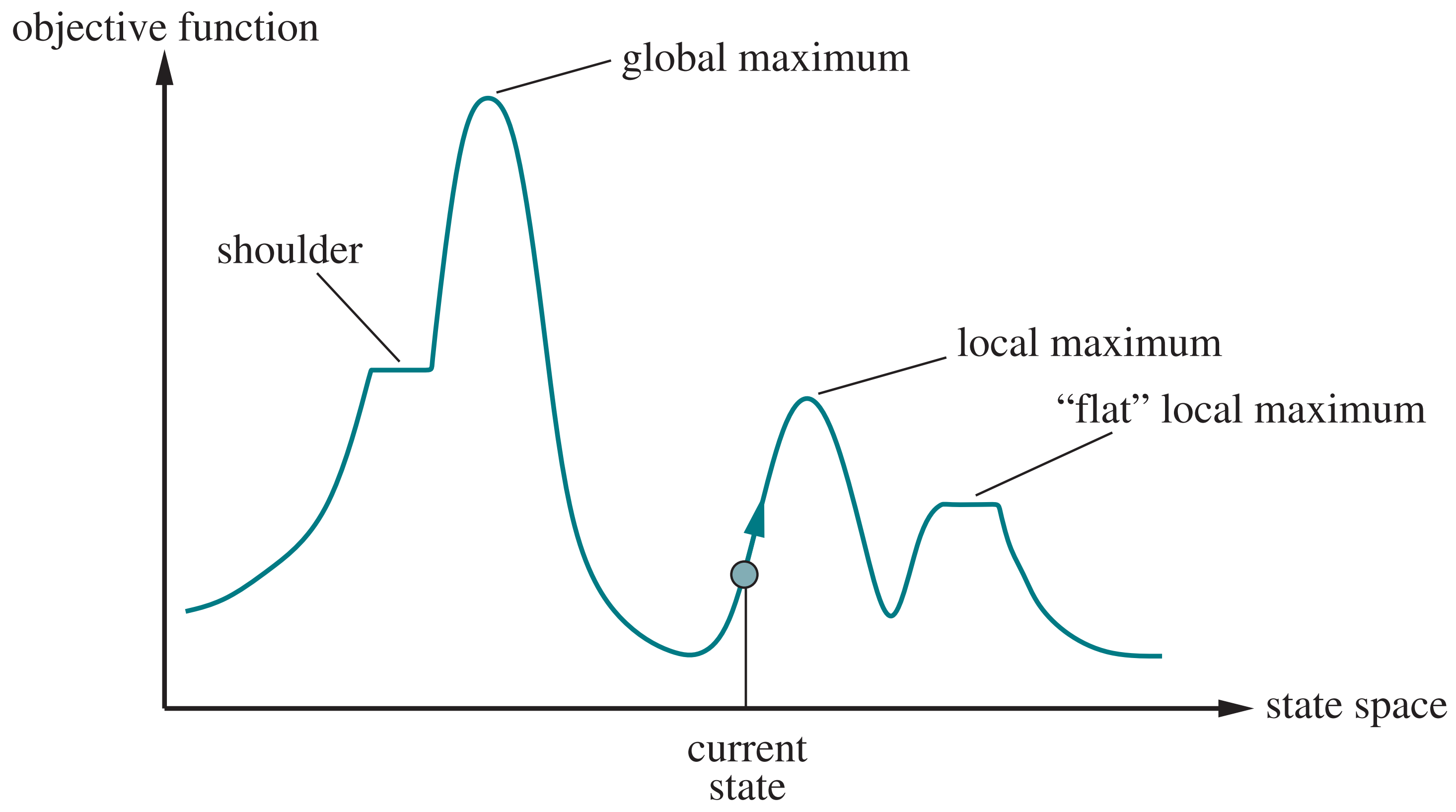
For such problems we can try **local search** techniques.

- we keep a single state only (constant memory)
- in each step we slightly modify the state
- usually, the path is not stored
- the method can also look for an **optimal state**, where the optimality is defined by an objective function (defined for states).
  - For example, for the 8-queens problem the objective function can be defined by the number of conflicting queens – this is extra information about the quality of states.



Local search - terminology

Local search can be seen as a move in the state-space **landscape**, where coordinates define the state and elevation corresponds to the objective function.



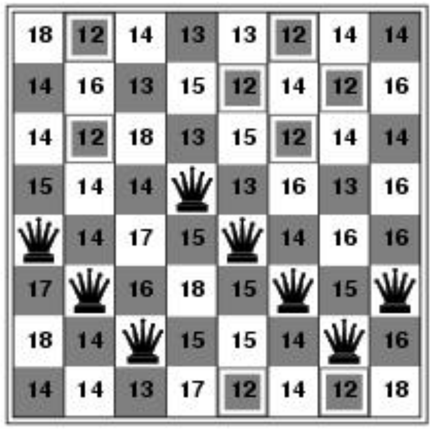
Hill climbing

From the neighbourhood the algorithm selects the state with the best value of the objective function and moves there (**hill climbing**).

- knows only the neighbourhood
- the current state is forgotten

- **# conflicts = 17**
- **state change = change row of a queen**
- **random selection among more best neighbours**

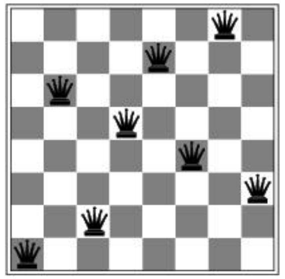
```
function HILL-CLIMBING($problem$) returns a state that is a local maximum
    $current \leftarrow problem$.INITIAL
    while $true$ do
        $neighbor \leftarrow$ a highest-valued successor state of $current$
        if VALUE($neighbor$) $\leq$ VALUE($current$) then return $current$
        $current \leftarrow neighbor$
```



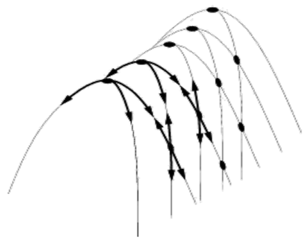
Hill climbing: problems

HC is a **greedy algorithm** – goes to the best neighbour without looking ahead

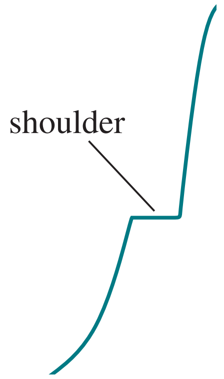
- **local optimum** (the state such that no neighbour is better)
  - HC cannot escape local optimum



- **ridges** (a sequence of local optima)
  - difficult for greedy algorithms to navigate



- **plateaux** (a flat area of the state-space landscape)
  - shoulder – progress is still possible
  - HC may not find a solution (cycling)



Hill climbing - versions

stochastic HC

- chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move
- usually converges more slowly than steepest ascent
- in some landscapes, it finds better solutions

first-choice HC

- implements stochastic HC until a successor better than the current state is generated
- a good strategy when a state has many (thousands) of successors

random-restart HC

- conducts a series of HC searches from randomly generated initial states (restart)
- can escape from a local optimum
- if HC has a probability p of success then the expected number of restarts required is $1/p$
- a very efficient method for the N-queens problem ($p \approx 0.14$, i.e., 7 iterations to find a goal)

Simulated annealing

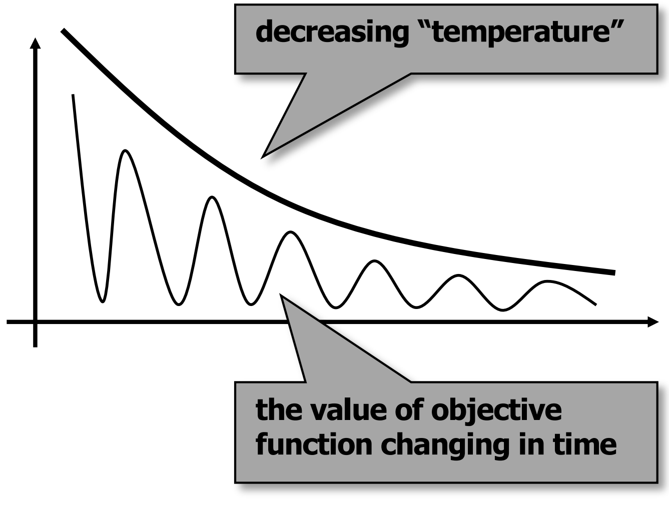
HC never makes the “downhill” moves towards states with lower value so the algorithm is not complete (can get stuck on a local optimum).

Random walk – that is moving to a successor chosen randomly – is complete but extremely inefficient.

Simulated annealing combines hill climbing with random walk

- motivation in metallurgy – process to harden metals by heating them to a high temperature and then gradually cooling them (allowing the material to reach a low-energy crystalline state)
- the algorithm picks a random move and accepts it if:
  - it improves the situation
  - it worsens the situation but this is allowed with a probability given by some temperature value and how much the state worsens; the temperature is decreasing according to a **cooling scheme**

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



Local beam search

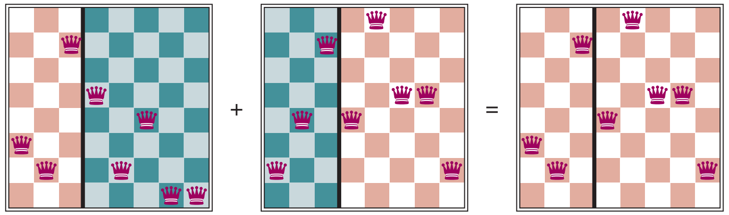
Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations.

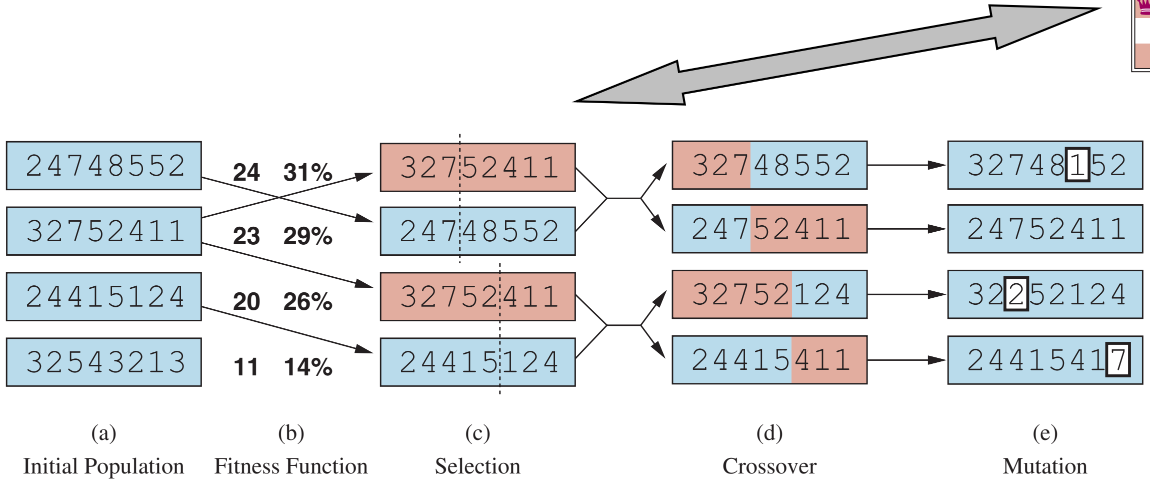
Can we exploit available memory better?

- **Local beam search** algorithm
  - keeps track of *k* states rather than just one
  - at each step, all the successors of all *k* states are generated
    - if any one is a goal, the algorithm halts
  - otherwise, it selects the *k* best successors and repeats
- **This is not running *k* restarts of HC in parallel!**
  - useful information is passed among the parallel search threads
  - the algorithm quickly abandons unfruitful searches and moves its resources to where the most is being made
  - can suffer from a lack of diversity
    - stochastic beam search helps alleviate this problem (*k* successors are chosen at random with the probability being an increasing function of state value)
    - resemble the process of natural selection

A variant of stochastic beam search in which successors are generated by combing two parent states (sexual reproduction)

- begin with a set of *k* randomly generated states – **population**
  - each state is represented as a string over a finite alphabet (DNA)
  - **fitness** function evaluates the states (objective function)
- select a pair of states for reproduction (probability of selection is given by the fitness function)
- for each pair choose a **crossover** point from the positions in the string
- combine **offsprings** to a new state
- each location is subject to random **mutation** with a small probability





Genetic algorithm

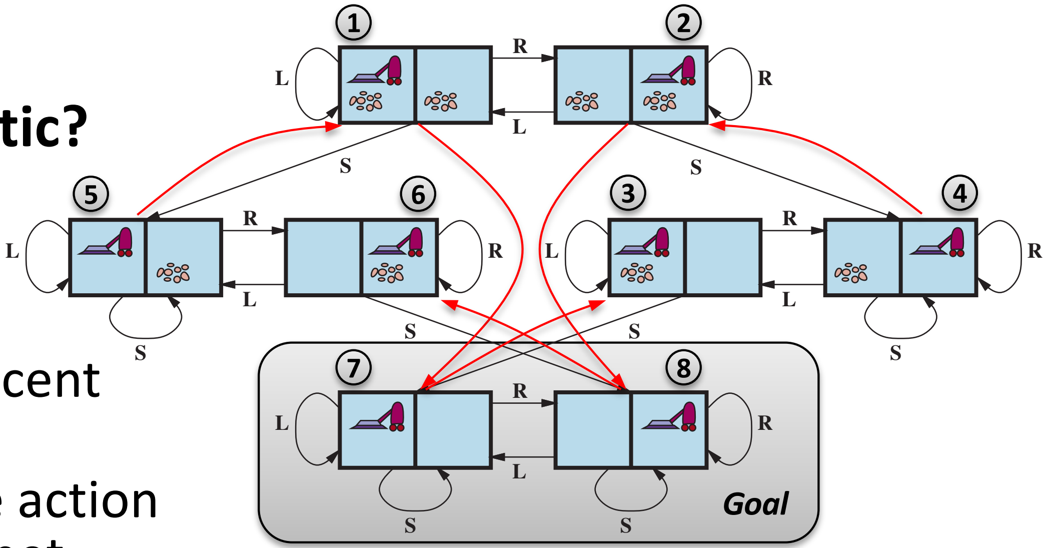
```
function GENETIC-ALGORITHM($population, fitness$) returns an individual
  repeat
      $weights \leftarrow$ WEIGHTED-BY($population, fitness$)
      $population2 \leftarrow$ empty list
      for $i$ = 1 to SIZE($population$) do
          $parent1, parent2 \leftarrow$ WEIGHTED-RANDOM-CHOICES($population, weights, 2$)
          $child \leftarrow$ REPRODUCE($parent1, parent2$)
          if (small random probability) then $child \leftarrow$ MUTATE($child$)
          add $child$ to $population2$
      $population \leftarrow population2$
  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
  $n \leftarrow$ LENGTH($parent1$)
  $c \leftarrow$ random number from 1 to $n$
  return APPEND(SUBSTRING($parent1, 1, c$), SUBSTRING($parent2, c+1, n$))
```

The erratic vacuum world

Recall the vacuum word with the cleaning robot that can **move right**, **move left**, or **suck**.

To reach the goal state from state 1, a possible (optimal) plan could be:

[Suck, Right, Suck]



What if the vacuum cleaner is erratic?

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.

We need to modify the **transition model** to include non-determinism of actions Suck:

Result(1, Suck) = {5,7}

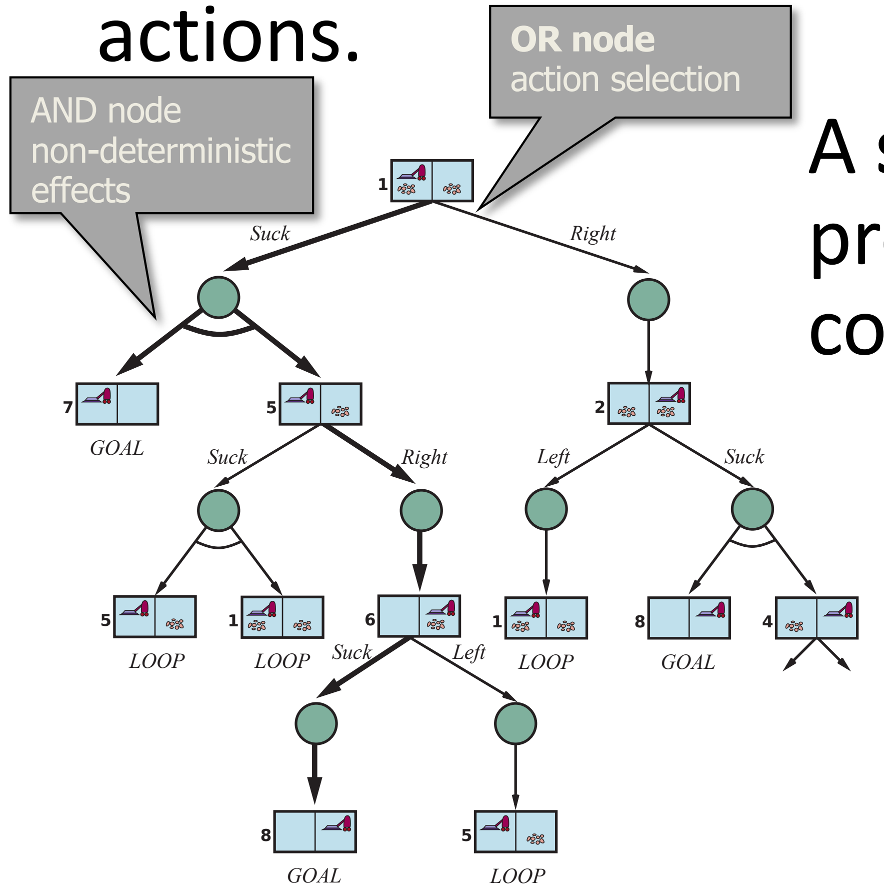
No single sequence of actions solves the problem, but the **conditional plan** does:

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

AND-OR search tree

In the classical search tree, the branching describes choices of the agent – **OR nodes**.

We add new type of nodes to the search tree, **AND nodes**, describing non-deterministic outcomes of actions.



A **solution** to AND-OR search problem is a subtree of the complete search tree that

- has a goal node at every leaf
- specifies one action at each of its OR nodes
- includes every outcome branch of its AND nodes

Depth-first search algorithm for AND-OR graph search

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

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

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

Looking for noncyclic solutions

Selecting the action

Covering all non-deterministic effects

Offline vs. online

So far we have concentrated on **offline search**

- compute a complete solution
- then execute the solution without assuming percepts

Online search is different in

- interleave computing and acting
  - select an action
  - execute an action
  - observe the environment
  - compute the next action
- a good idea in dynamic and semidynamic environments
- helpful in nondeterministic domains (unknown actions and unknown results of actions)

Online search is useful **for agents executing actions** (useless for pure computation).

The agent knows only the following **information**:

– **Actions(s)** – a list of actions allowed in state *s*

– **c(s,a,s‘)** – the step cost function (cannot be used until the agent knows state s‘)

– **Goal-Test(s)** – identifying the goal state

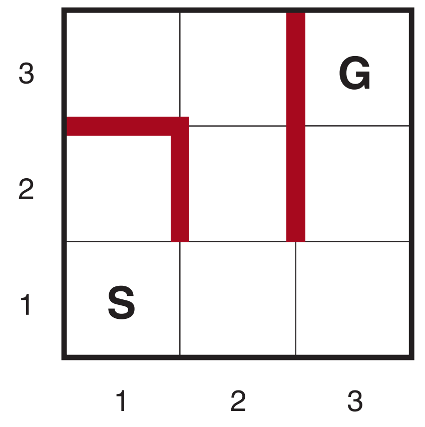
We assume the following:

– agent can **recognize a visited state**

– (agent can build a world map)

– actions are deterministic

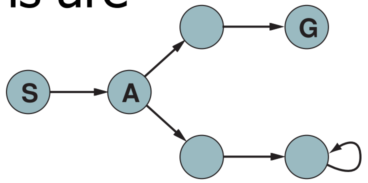
– agent has an **admissible heuristic** h(s)



Evaluating on-line algorithms

Quality of online algorithms can be measured by **comparing with the offline solution** (knowing the best solution in advance).

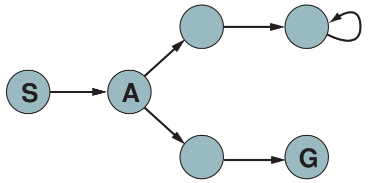
- **Competitive ratio**
  = quality of the online solution / quality of the best solution
  - can be ∞, for example for a **dead-end** state (if some actions are **irreversible**).



Claim: No algorithm can avoid dead ends in all state spaces.

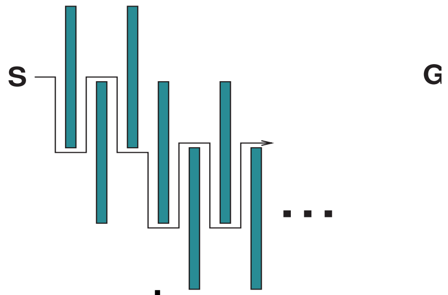
Proof (adversary argument)

Agent has visited states S and A must make the same decision in both, but in one situation, the agent reaches dead-end.



Assume that the state space is **safely explorable** (some goal state is reachable from every reachable state).

- No bounded competitive ratio can be guaranteed if there are paths of unbounded cost.
- Adversary argument can be used to arbitrarily extend any path.



Hence it is common to describe the performance of online search algorithms in **terms of the size of the entire state space** rather than just the depth of the shallowest goal.

Online DFS

Opposite to offline algorithms such as A* online algorithms can discover **successors only** for a node that the agent physically occupies.

It seems better to expand nodes in a **local order** as done for example by DFS.

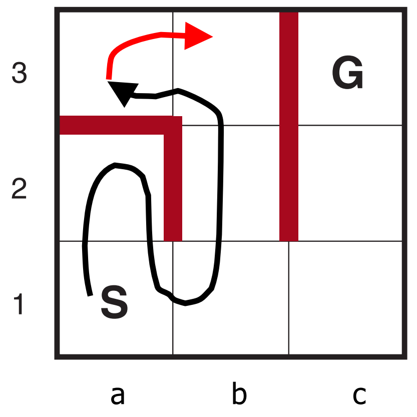
```
function ONLINE-DFS-AGENT($problem$, $s'$) returns an action
                  $s$, $a$, the previous state and action, initially null
    persistent: $result$, a table mapping $(s, a)$ to $s'$, initially empty
                  $untried$, a table mapping $s$ to a list of untried actions
                  $unbacktracked$, 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 $stop$
        else $a \leftarrow$ an action $b$ such that $result[s', b]$ = POP($unbacktracked[s']$) ; $s' \leftarrow null$
    else $a \leftarrow$ POP($untried[s']$)
    $s \leftarrow s'$
    return $a$
```

learns outcome of actions

The state can be visited multiple times in a single journey (it is leaved to different states) so we need to remember where to go back – **Ariadne's thread**

to go back we need a reverse action to the actions used to reach the state

Online DFS

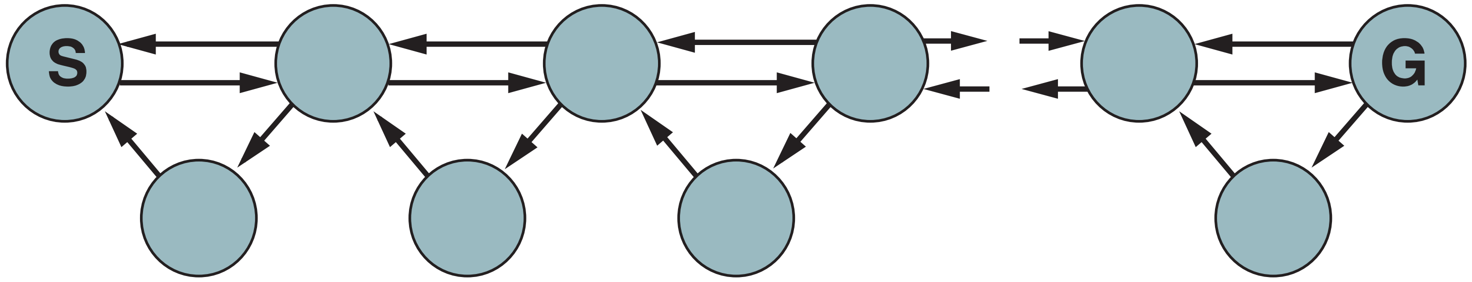


state	unEX	unBT	rUP	rDN	rLF	rRG
(1,a)	{}	(2,a)	(2,a)	-	-	(1,b)
(2,a)	{}	(1,a)	-	(1,a)	-	-
(1,b)	LF,RG	(1,a)	(2,b)	-		
(2,b)	DW	(1,b)	(3,b)		-	-
(3,b)	DW	(3,a), (2,b)	-		(3,a)	-
(3,a)		(3,b)	-	-	-	(3,b)

- In the worst case, every link is traversed exactly twice (forward and backward).
- This is optimal for exploration, but for finding a goal, the agent‘s competitive ratio could be arbitrarily bad.
  - an online variant of iterative deepening solves this problem
- On-line DFS works only in state spaces where **actions are reversible**.

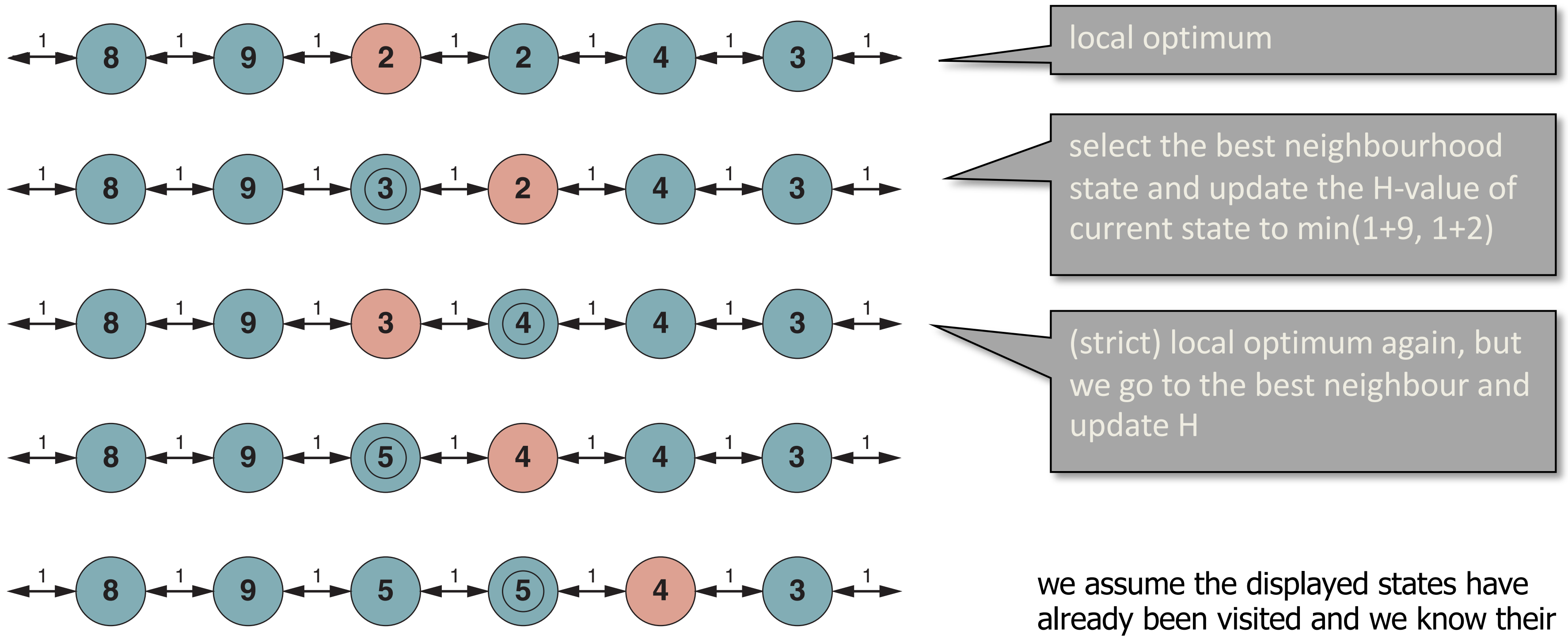
Hill climbing is an on-line algorithm.

- keeps a **single state** (the current physical state)
- does **local steps** to the neighbouring states
- in its simplest form **cannot escape local optima**
  - Beware! **Restarts cannot be used in online search!**
  - We can still use **random walk**.
    - A random walk will **eventually find a goal** or will complete its exploration, provided that the space is finite.
    - The process **can be very slow**.
    - In the following example, a random walk will take exponentially many steps to find the goal (at each step, backward progress is twice as likely as forward progress).



We can exploit available memory to remember visited states and hence leave local optima.

– **H(s)** – the current best estimate of path length from *s* to the goal (equals h(s) at the beginning)



1
8
9
2
2
4
3
3
5
4
local optimum
select the best neighbourhood state and update the H-value of current state to min(1+9, 1+2)
(strict) local optimum again, but we go to the best neighbour and update H

we assume the displayed states have already been visited and we know their H-values

Learning real-time A*

Algorithm LRTA* makes local steps and learns the result of each action as well as a better estimate of distance to the goal (H).

```
**function** LRTA*-AGENT($problem,\ s',\ h$) **returns** an action
                  $s,\ a$, the previous state and action, initially null
    **persistent**: $result$, 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**
        $result[s, a] \leftarrow s'$
        $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(s, b, result[s, b], H)$
    $a \leftarrow \operatorname{argmin}_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(problem, s', b, result[s', b], H)$
    $s \leftarrow s'$
    **return** $a$

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

improve the H-function in the previous state

select the next action with the best cost (can also go back); prefer not-yet explored states

if the action has not been applied yet, we optimistically assume that it leads to state with the best cost, i.e. h(s)

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz