# Beyond Classical Search

(Local Search)

R&N III: Chapter 4



## Local Search

- Light-memory search method
- No search tree; only the current state is represented!
- Only applicable to problems where the path is irrelevant (e.g., 8-queen), unless the path is encoded in the state
- Many similarities with optimization techniques



## Hill-climbing search

- "is a loop that continuously moves in the direction of increasing value"
  - It terminates when a peak is reached.
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Basic Hill-climbing – "Like climbing Everest in a thick fog with amnesia"



## (Steepest Ascent) Hill-climbing search

- "is a loop that continuously moves in the direction of increasing value"
  - It terminates when a peak is reached.
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Hill-climbing chooses randomly among the set of best successors, if there is more than one.
- Hill-climbing a.k.a. *greedy local search*



## Steepest Ascent Hill-climbing search

```
function HILL-CLIMBING( problem) return a state that is a local maximum
    input: 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
```



## Hill-climbing example

- 8-queens problem (complete-state formulation).
- Successor function: move a single queen to another square in the same column.
- Heuristic function *h(n)*: the number of pairs of queens that are attacking each other (directly or indirectly).

## Hill-climbing example

a) b)

a) shows a state of h=17 and the h-value for each possible successor.

b) A local minimum in the 8-queens state space (h=1).



## Drawbacks

- Ridge = sequence of local maxima difficult for greedy algorithms to navigate
- Plateaux = an area of the state space where the evaluation function is flat.
- Gets stuck 86% of the time.



## Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
  - cfr. stochastic hill climbing by generating successors randomly until a better one is found.
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.



## Steepest Descent

1) S ← initial state
2) Repeat:
   a) S' ← arg min$_{S' \in SUCCESSORS(S)}$ {h(S')}
   b) if GOAL?(S') return S'
   c) if h(S') < h(S) then S ← S' else return failure

Similar to:
- hill climbing with –h
- gradient descent over continuous space



## Random Restart Application: 8-Queen

Repeat n times:
1) Pick an initial state S at random with one queen in each column
2) Repeat k times:
   a) If GOAL?(S) then return S
   b) Pick an attacked queen Q at random
   c) Move Q it in its column to minimize the number of attacking queens is minimum → new S [min-conflicts heuristic]
3) Return failure

## Application: 8-Queen

**Why does it work ???**

1) There are **many** goal states that are well-distributed over the state space
2) If no solution has been found after a few steps, it's better to start it all over again. Building a search tree would be much less efficient because of the high branching factor
3) Running time almost independent of the number of queens

## Steepest Descent

1) S ← initial state
2) Repeat:
   a) S' ← arg min$_{S' \in \text{SUCCESSORS}(S)}$ {h(S')}
   b) if GOAL?(S') return S'
   c) if h(S') < h(S) then S ← S' else return failure

may easily get stuck in local minima

→ Random restart (as in n-queen example)

→ Monte Carlo descent



## Simulated annealing

- Escape local maxima by allowing "bad" moves.
  - Idea: but gradually decrease their size and frequency.
- Origin; metallurgical annealing
- Bouncing ball analogy:
  - Shaking hard (= high temperature).
  - Shaking less (= lower the temperature).
- If T decreases slowly enough, best state is reached.
- Applied for VLSI layout, airline scheduling, etc.



## Simulated annealing

**function** SIMULATED-ANNEALING( *problem, schedule*) **return** a solution state
  **input:** *problem*, a problem
    *schedule*, a mapping from time to temperature
  **local variables:** *current*, a node.
    *next*, a node.
    *T*, a "temperature" controlling the probability 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*
    ∆*E* ← VALUE[*next*] - VALUE[*current*]
    **if** ∆*E* > 0 **then** *current* ← *next*
    **else** *current* ← *next* only with probability $e^{\Delta E/T}$



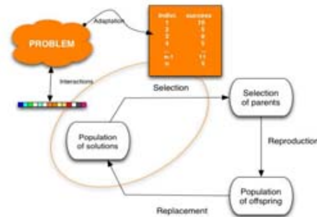
## Local beam search

- Keep track of *k* states instead of one
  - Initially: *k* random states
  - Next: determine all successors of *k* states
  - If any of successors is goal → finished
  - Else select *k* best from successors and repeat.
- Major difference with random-restart search
  - Information is shared among *k* search threads.
- Can suffer from lack of diversity.
  - Stochastic variant: choose k successors at proportionally to state success.



## Genetic algorithms

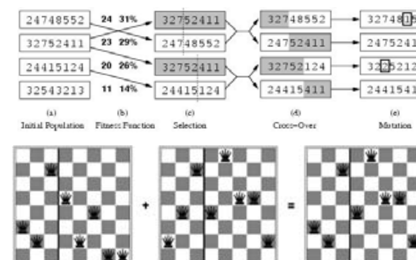
- Variant of local beam search with *sexual recombination.*






## Genetic algorithms

- Variant of local beam search with *sexual recombination.*



# Genetic algorithm

```
function GENETIC_ALGORITHM( population, FITNESS-FN) return an individual
  input: population, a set of individuals
         FITNESS-FN, a function which determines the quality of the individual
  repeat
         new_population ← empty set
         loop for i from 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
```



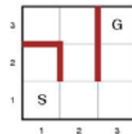
# Exploration problems

- Until now all algorithms were offline.
  - Offline= solution is determined before executing it.
  - Online = interleaving computation and action
- Online search is necessary for dynamic and semi-dynamic environments
  - It is impossible to take into account all possible contingencies.
- Used for *exploration problems*:
  - Unknown states and actions.
  - e.g. any robot in a new environment, a newborn baby,…



# Online search problems

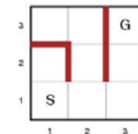
- Agent knowledge:
  - ACTION(s): list of allowed actions in state s
  - C(s,a,s’): step-cost function (! After s’ is determined)
  - GOAL-TEST(s)
- An agent can recognize previous states.
- Actions are deterministic.
- Access to admissible heuristic *h(s)*
  e.g. manhattan distance





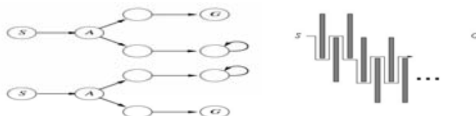

# Online search problems

- Objective: reach goal with minimal cost
  - Cost = total cost of travelled path
  - Competitive ratio=comparison of cost with cost of the solution path if search space is known.
  - Can be infinite in case of the agent accidentally reaches dead ends






# The adversary argument



- Assume an adversary who can construct the state space while the agent explores it
  - Visited states S and A. What next?
    - Fails in one of the state spaces
- No algorithm can avoid dead ends in all state spaces.



# Online search agents

- The agent maintains a map of the environment.
  - Updated based on percept input.
  - This map is used to decide next action.

Note difference with e.g. A*

An online version can only expand the node it is physically in (local order)

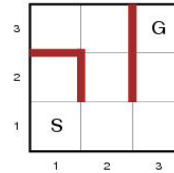
# Online DF-search

```
function ONLINE_DFS-AGENT(s') return an action
    input: s', a percept identifying current state
    static: result, a table indexed by action and state, initially empty
            unexplored, a table that lists for each visited state, the action not yet tried
            unbacktracked, a table that lists for each visited state, the backtrack 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 then unexplored[s'] ← ACTIONS(s')
    if s is not null then do
            result[a,s] ←  s'
            add s to the front of unbackedtracked[s']
    if unexplored[s'] is empty then
            if unbacktracked[s'] is empty then return stop
            else a ← an action b such that result[b, s']=POP(unbacktracked[s'])
    else a ← POP(unexplored[s'])
    s ←  s'
    return a
```



# Online DF-search, example

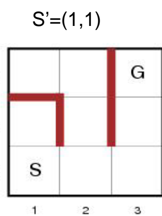



- Assume maze problem on 3x3 grid.
- s' = (1,1) is initial state
- Result, unexplored (UX), unbacktracked (UB), …
  are empty
- S,a are also empty



# Online DF-search, example

S'=(1,1)

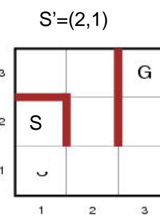



- GOAL-TEST((,1,1))?
  - S not = G thus false
- (1,1) a new state?
  - True
  - ACTION((1,1)) -> UX[(1,1)]
    - {RIGHT,UP}
- s is null?
  - True (initially)
- UX[(1,1)] empty?
  - False
- POP(UX[(1,1)])->a
  - A=UP
- s = (1,1)
- Return a



# Online DF-search, example

S'=(2,1)

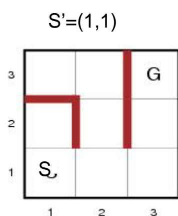



- GOAL-TEST((2,1))?
  - S not = G thus false
- (2,1) a new state?
  - True
  - ACTION((2,1)) -> UX[(2,1)]
    - {DOWN}
- s is null?
  - false (s=(1,1))
  - result[UP,(1,1)] <- (2,1)
  - UB[(2,1)]={(1,1)}
- UX[(2,1)] empty?
  - False
- A=DOWN, s=(2,1) return A



# Online DF-search, example

S'=(1,1)

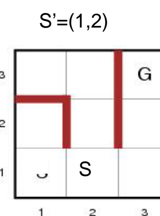



- GOAL-TEST((1,1))?
  - S not = G thus false
- (1,1) a new state?
  - false
- s is null?
  - false (s=(2,1))
  - result[DOWN,(2,1)] <- (1,1)
  - UB[(1,1)]={(2,1)}
- UX[(1,1)] empty?
  - False
- A=RIGHT, s=(1,1) return A



# Online DF-search, example

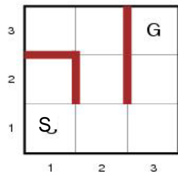
S'=(1,2)




- GOAL-TEST((1,2))?
  - S not = G thus false
- (1,2) a new state?
  - True, UX[(1,2)]={RIGHT,UP,LEFT}
- s is null?
  - false (s=(1,1))
  - result[RIGHT,(1,1)] <- (1,2)
  - UB[(1,2)]={(1,1)}
- UX[(1,2)] empty?
  - False
- A=LEFT, s=(1,2) return A

## Online DF-search, example

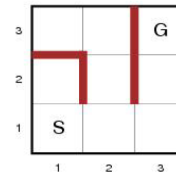
S’=(1,1)




- GOAL-TEST((1,1))?
  - S not = G thus false
- (1,1) a new state?
  - false
- s is null?
  - false (s=(1,2))
  - result[LEFT,(1,2)] <- (1,1)
  - UB[(1,1)]={(1,2),(2,1)}
- UX[(1,1)] empty?
  - True
  - UB[(1,1)] empty? False
- A= b for b in result[b,(1,1)]=(1,2)
  - B=RIGHT
- A=RIGHT, s=(1,1) …



## Online DF-search

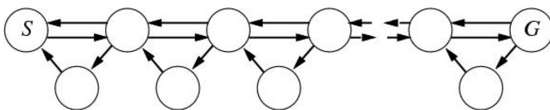



- Worst case each node is visited twice.
- An agent can go on a long walk even when it is close to the solution.
- An online iterative deepening approach solves this problem.
- Online DF-search works only when actions are reversible.



## Online local search

- Hill-climbing is already online
  - One state is stored.
- Bad performance due to local maxima
  - Random restarts impossible.
- Solution: Random walk introduces exploration (can produce exponentially many steps)






## Online local search

- Solution 2: Add memory to hill climber
  - Store current best estimate *H(s)* of cost to reach goal
  - *H(s)* is initially the heuristic estimate *h(s)*
  - Afterward updated with experience (see below)
- Learning real-time A* (LRTA*)