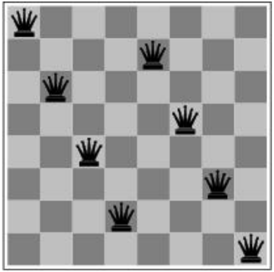
Search in Complex Environments

Jihoon Yang

Machine Learning Research Laboratory
Department of Computer Science & Engineering
Sogang University

Local search and optimization

- Previously: systematic exploration of search space
  - Solution to problem is path to goal
- Local search: evaluate and modify one or more current states; All that matters is the solution state, not the path cost to reach it
- E.g. 8-queens using complete-state formulation
  - States: 8 queens on the board, one per column
  - Actions: Move a queen to a square in the same column



Local search and optimization

- Local search is suitable for **optimization problems**
  - State space = set of "complete" configurations
  - Find best state according to some *objective function* $h(s)$
  - E.g. $h(s)$ =number of conflicts

- Many optimization problems do not fit the standard search model

- Local search = keep a single current state and move to neighboring states to improve it

- Advantages:
  - Use very little memory
  - Often find reasonable solutions in large or infinite state spaces unsuitable for systematic algorithms (e.g. solve million-queens quickly)

Hill-climbing search

- Keep a single current node and move to neighboring states to improve it
- A loop that continuously moves in the direction of increasing value
  - Chooses randomly to break ties
  - It terminates when a peak is reached where no neighbor has a higher value
  - "Like climbing Everest in thick fog with amnesia"
- Hill-climbing a.k.a. *greedy local search*, *steepest ascent/descent*

Hill-climbing search

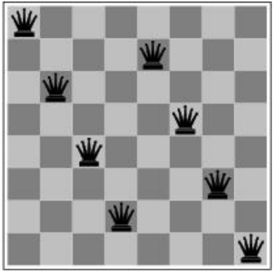
function HILL-CLIMBING(*problem*) **return** a state that is a local maximum

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*

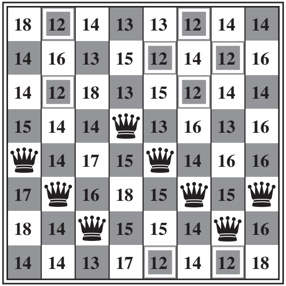
Hill-climbing example

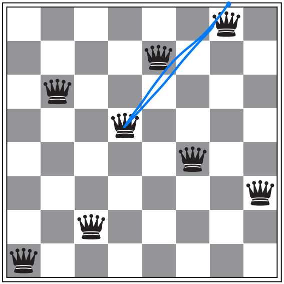
- E.g. 8-queens (complete-state formulation)
  - States: 8 queens on the board, one per column
  - Actions: Move a single queen to another square in the same column

- Heuristic function $h(n)$: the number of pairs of queens that are attacking each other (directly or indirectly)



Hill-climbing example

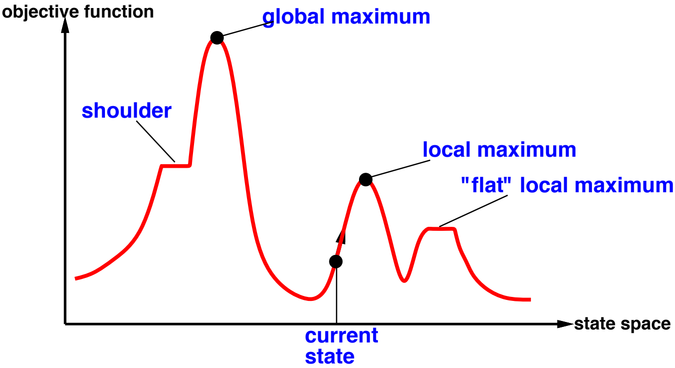




- The first shows a state of $h = 17$ and the h-value for each possible successor
- The second shows a local minimum in the 8-queens state space $(h = 1)$

Drawbacks

State space landscape



- Depending on initial state, can get stuck in local maxima, plateaux
- A *sideways move* can be allowed in the hope that the plateau is really a shoulder

Hill-climbing variations

- Stochastic HC
  - Random selection among the uphill moves
  - The selection probability can vary with the steepness of the uphill move
- First-choice HC
  - Generating successors randomly until a better (than the current) one is found
  - Useful when a state has many successors
- Random-restart HC
  - Restart search from random initial state
  - A reasonably good local maximum can often be found after a small number of restarts

Simulated annealing

- Escape local maxima by allowing "bad" moves
  - Idea: but gradually decrease their frequency
- T, a "temperature" controlling the probability of downward steps
- One can prove: If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Commonly used: $T \leftarrow cT$ with c constant close to, but smaller than, 1
- Applied to 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: *T***,** a "temperature" controlling the probability of downward steps

current $\leftarrow$ MAKE-NODE(problem.INITIAL-STATE)

for t $\leftarrow$ **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 $e^{\Delta E / T}$

T는 항상 양수

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 $\rightarrow$ 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 randomly with probability proportional to state value difference

Genetic algorithms

- Inspired by the process of biological evolution
- Start with k randomly generated states/individuals (**population**)
- States are scored by evaluation function (**fitness function**)
- At each step, the most fit states are selected (*survival of the fittest*) probabilistically: used as seeds for producing the next generation population – the children or *offspring*, by means of operations such as **crossover** and **mutation**
- The process is repeated until sufficiently fit states are discovered – the best state has a score exceeding a criterion
- They have been applied successfully to a variety of learning tasks and optimization problems

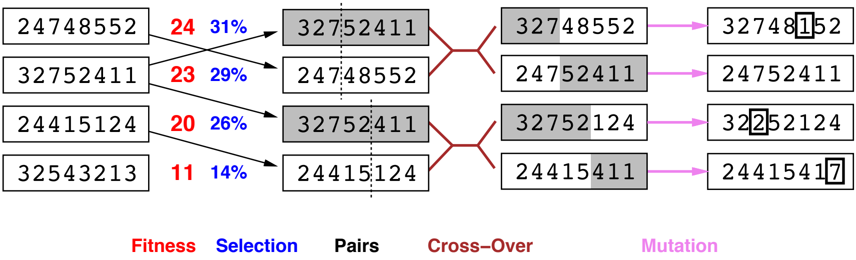
Genetic algorithms

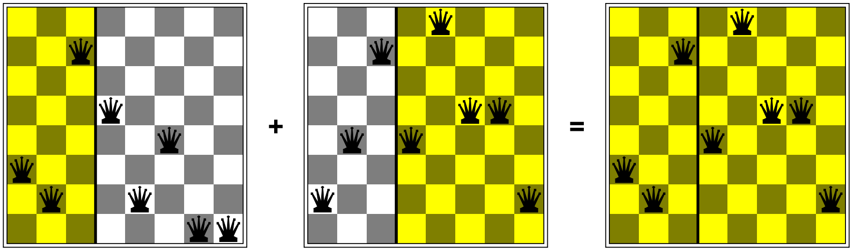
```
function GENETIC_ALGORITHM(population, FITNESS-FN) return an
  individual
  input: population, a set of individuals
         FITNESS-FN, a function which determines the fitness of an
         individual
  repeat
         new_population ← empty set
         loop for i from 1 to SIZE(population) do
              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 in population
```

Genetic algorithms

- In basic GAs, the fundamental representation of each state is a *string* over a finite alphabet (often a string of 0s and 1s), called a **chromosome**
- The mapping depends on the problem domain, and the designers
- Genetic Operators
  - **Replication**: A chromosome is merely reproduced
  - **Crossover**: Involves the mating of two parent chromosomes to yield two new offspring by copying selected bits from each parent
  - **Mutation**: Creates a single descendant from a single parent by changing the value of a randomly chosen bit (from a 1 to 0 or vice versa)
  - Other operators: specialized to the particular representation

Genetic algorithms





Online search

Suppose we want to site three airports in Romania:

– 6-D state space defined by (x_1, y_2), (x_2, y_2), (x_3, y_3)

– objective function $f(x_1, y_2, x_2, y_2, x_3, y_3) =$

sum of squared distances from each city to nearest airport

Discretization methods turn continuous space into discrete space, e.g., empirical gradient considers $\pm\delta$ change in each coordinate

Gradient methods compute

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

to increase/reduce f, e.g., by $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

Sometimes can solve for $\nabla f(\mathbf{x}) = 0$ exactly (e.g., with one city).

Newton–Raphson (1664, 1690) iterates $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x})$ to solve $\nabla f(\mathbf{x}) = 0$, where $\mathbf{H}_{ij} = \partial^2 f / \partial x_i \partial x_j$

Online search

Offline search algorithms compute a complete solution before exec.

vs.

online search ones interleave computation and action (processing input data as they are received)

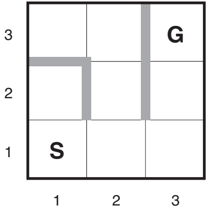
– necessary for unknown environment

(dynamic or semidynamic, and nonderterministic domains)

$\Leftarrow$ exploration problem

Online search

An online search agent solves problem by executing actions, rather than by pure computation (offline)



The competitive ratio – the total cost of the path that the agent actually travels (online cost) / that the agent would follow if it knew the search space in advance (offline cost) $\Leftarrow$ as small as possible

Online search expands nodes in a local order, say, DEPTHFIRST and HILLCLIMBING have exactly this property

Online search

```
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'] ← problem.ACTIONS(s')
  if s is not null then
      result[s, a] ← 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 ← an action b such that result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

Figure 4.20 An online search agent that uses depth-first exploration. The agent can safely explore only in state spaces in which every action can be "undone" by some other action.