Lecture 5: Beyond Classical Search

Artificial Intelligence

CS-6364

Outline

- *Local Search*
  - Hill-climbing search
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithms
- *On-Line Search*

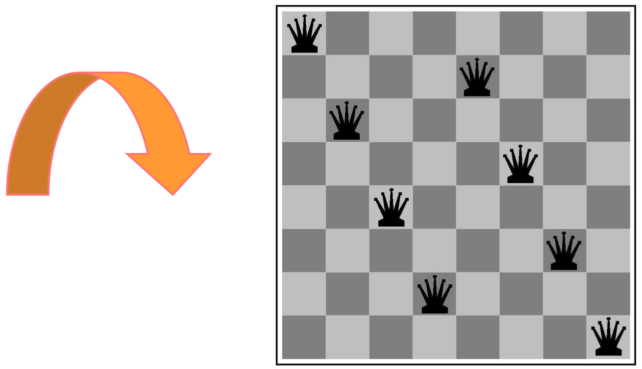
Local Search algorithms and optimization

- Until now we have studied systematic search algorithms
  - Start from an initial state
  - Keep one or more paths in memory
  - Record which alternatives have been explored at each point in the path
- Local search algorithms operate by using a single current state and generally move only to the neighbors of that state
  - The goal is irrelevant in many problems
  - The paths to the goal is not important
  - Non-systematic search, but (1) use little memory; (2) find reasonable solutions fast, even in large search spaces
- In addition to finding goals, local search algorithms solve optimizations problems
  - The aim is to find the best state according to an objective function

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

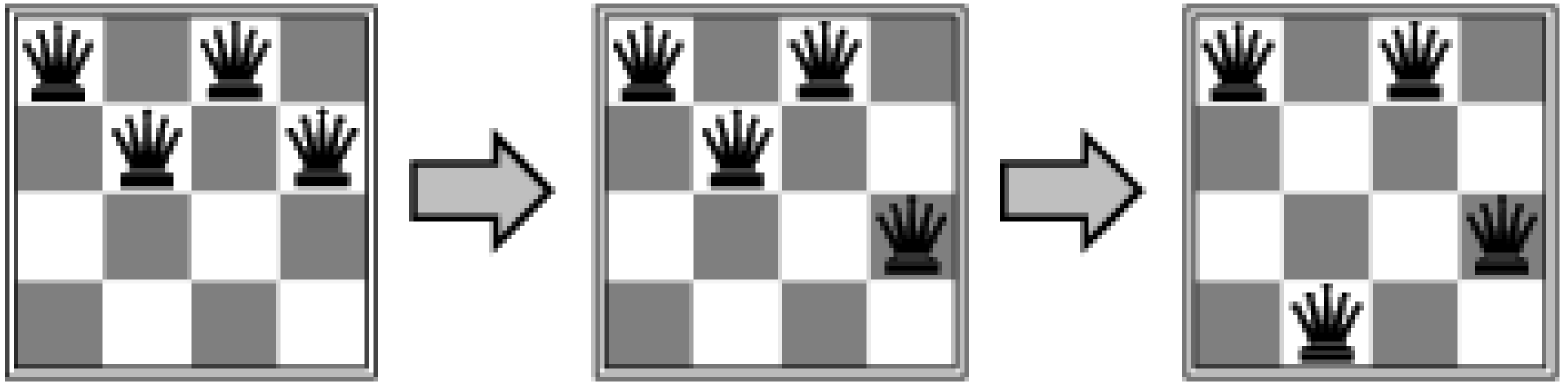
- State space = set of "complete" configurations
  - Find configuration satisfying constraints, e.g., n-queens



- In such cases, we can use local search algorithms
  - keep a single "current" state, try to improve it

Example: n-queens

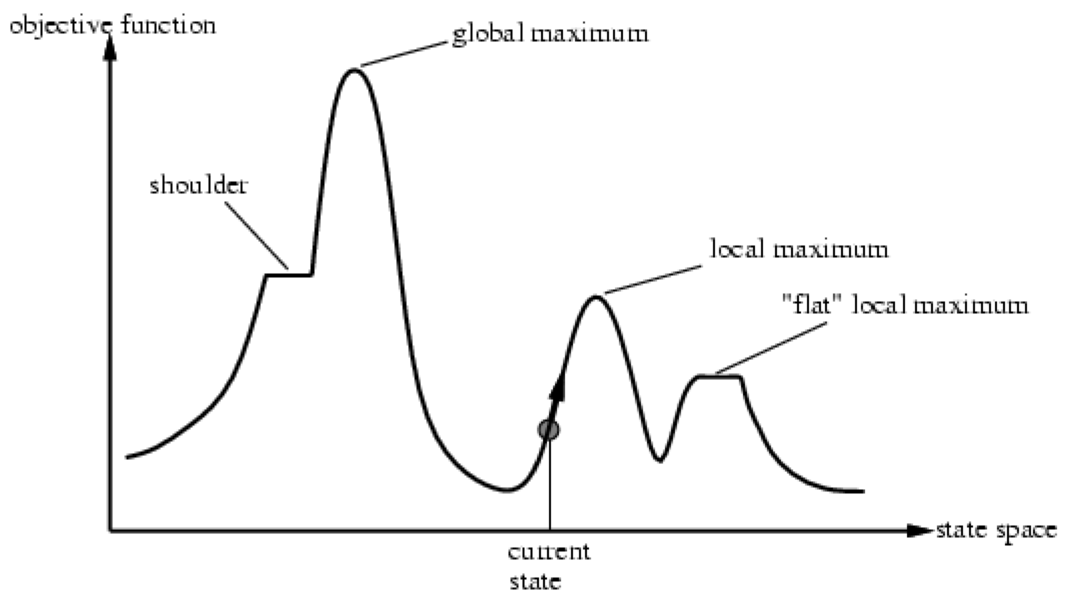
- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal



State space landscape

In order to understand local search, it is useful to consider the state-space landscape!

- It has "location" – defined by the state
- It has "elevation" – defined by the value of the heuristic cost or objective function

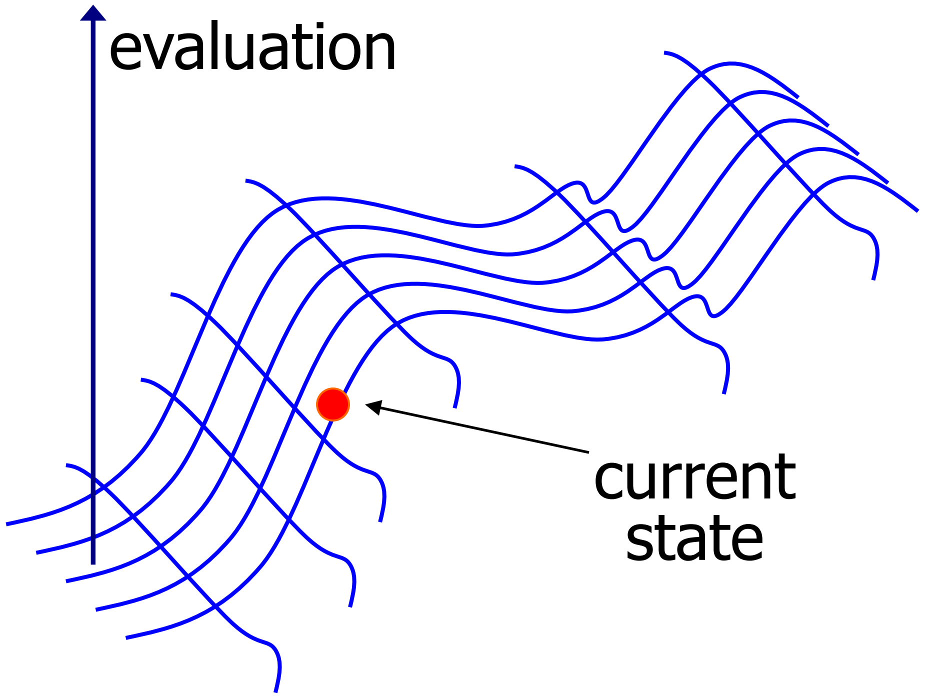


A complete local search algorithm always finds a Goal if one exists.

An optimal algorithm always finds the global minimum/maximum.

Iterative improvement algorithms

- Often the most practical approach
- Consider all states laid out on the surface of the landscape. The height corresponds to the evaluation function of the state=objective function in that state



- The idea of iterative improvement: move around the landscape trying to find the highest peaks = optimal solutions

Hill-Climbing Search

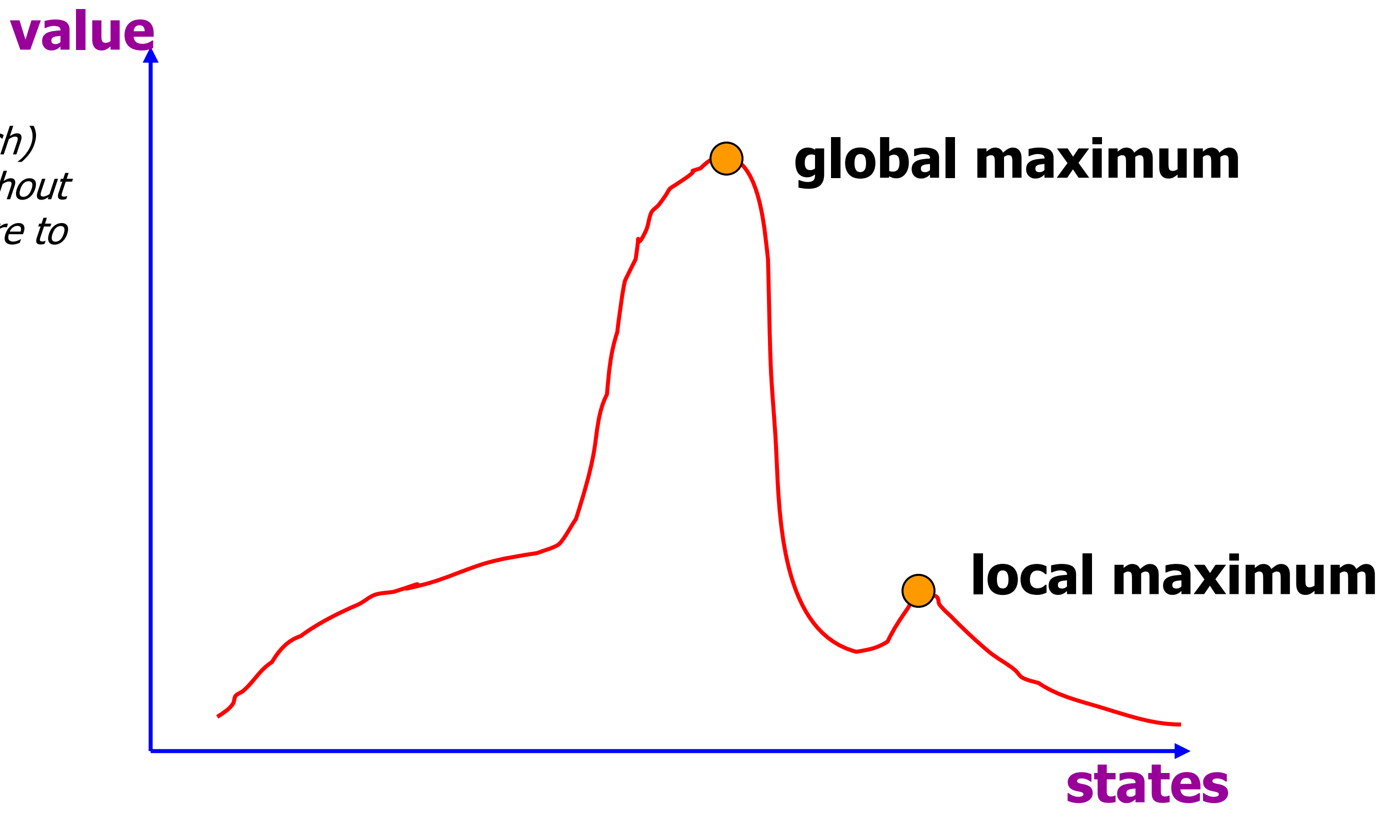
```
function HILL-CLIMBING ( problem ) returns a state that is a local maximum
  INPUTS:  problem, a problem
  LOCAL VARIABLES: current, a node
                   neighbor, a node

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

- Does not maintain a search tree!!!!
- Does not look ahead beyond the immediate neighbors of the current state!!

Hill climbing

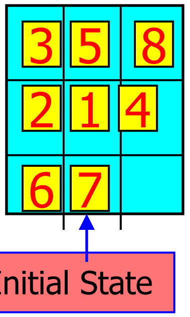
Problem: depending on initial state, can get stuck on local maxima

Hill-climbing (greedy search) grabs a good neighbor without thinking ahead about where to go next

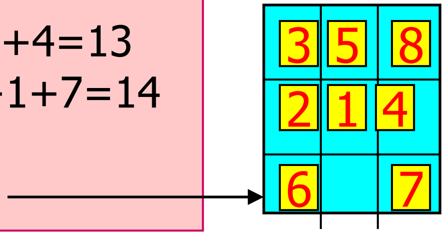


Example of Hill-Climbing Search

```
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 ← the highest valued successor of current
       if neighbor.VALUE ≤ current.VALUE then return current.STATE[
       current ← neighbor
```



Value(IS)=4+7=11
Value(successor$_{UP}$)=1+8+4=13
Value(successor$_{LEFT}$)=6+1+7=14
Neighbor=successor$_{LEFT}$
Current=Neighbor

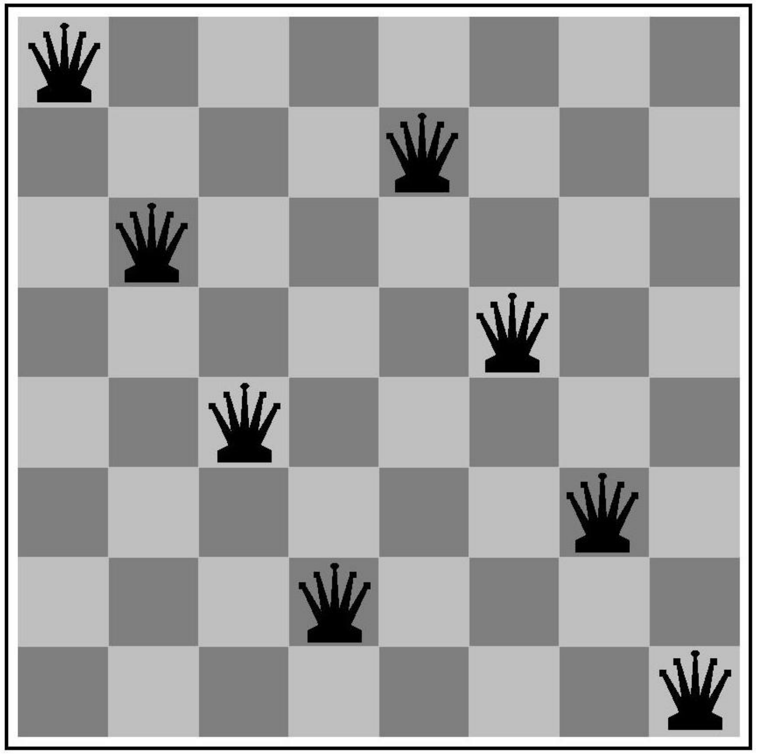


Value(state) = ΣValues(neighbor(empty))

Hill-climbing

- Hill-climbing search
  - always moves in the direction of increasing value. Because of this, it can get stuck in local maxima. To solve this problem, there is the <u>Random-Restart Hill Climbing</u> algorithm. This is the hill-climbing algorithm, run several times at different random locations. The assumption is that if you run the algorithm enough times, you will eventually find the best solution.

A problem



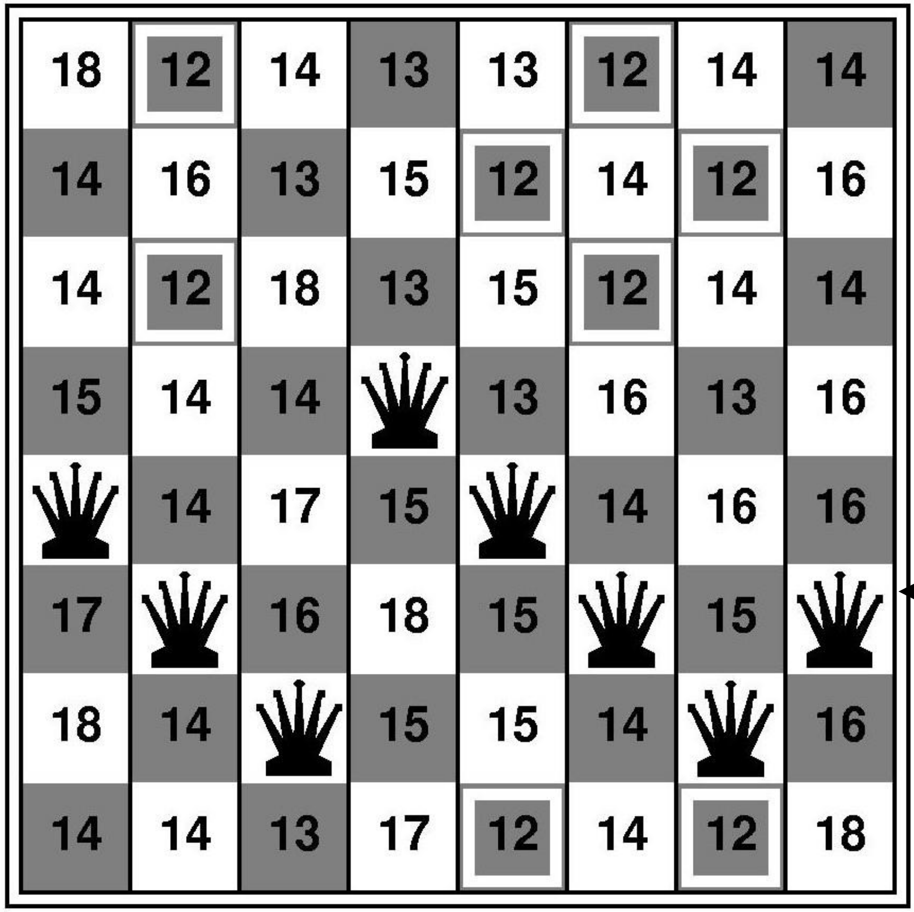
Place 8 queens on a chessboard
Such that no queen attacks the other

A queen attacks any piece on the
Same row, column or diagonal

Complete state formulation:
All queens are on the board, one
Per column

Successors??? *All possible states generated by moving a single queen*
to another square in the same column$\rightarrow$ each state has $7\times8=56$ succesors

Example: 8 queens



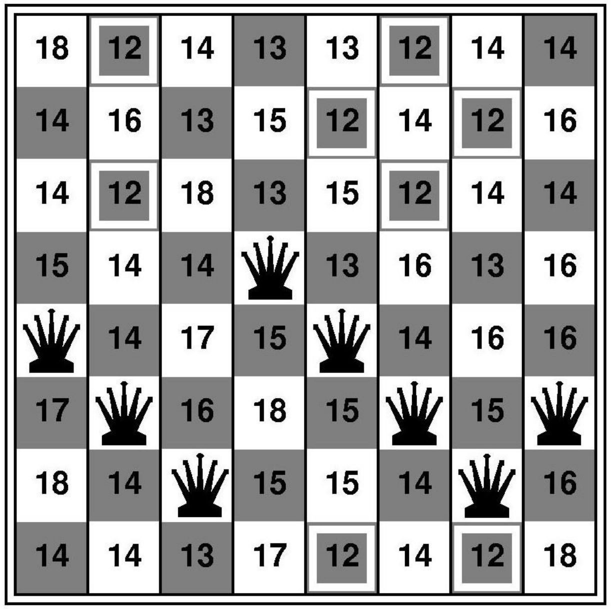
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

The heuristic cost:
# of pairs of queens that
Attack each other

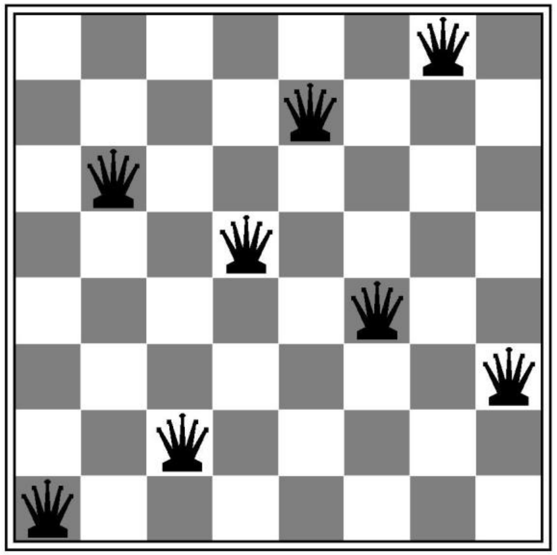
h = 17

How many successors?
64-8=56 or 8*7=56

Local minimum



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18



h = 17

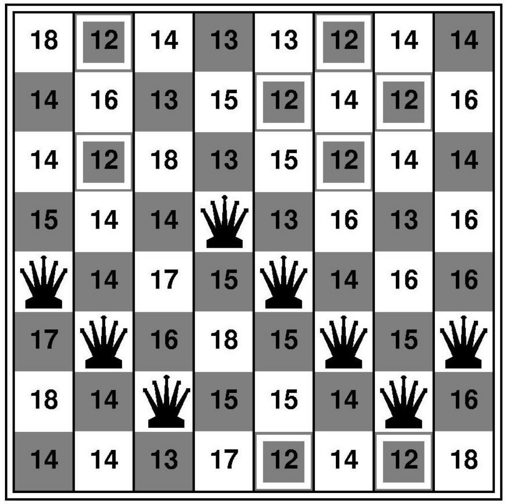
h = 1

Greedy Local Search

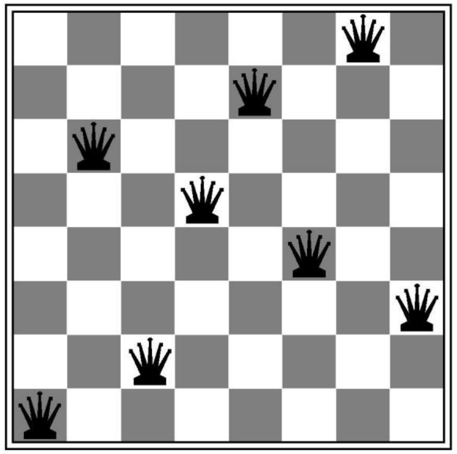
Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next. Although *greed* is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well.

Hill climbing often makes very rapid progress towards a solution, because it is usually quite easy to improve a bad state.

- For example, from the state in Figure(a), it takes just five steps to reach the state in Figure(b), which has $h=1$ and is very nearly a solution.



(a)



(b)

Hill climbing often gets stuck -3 reasons:

1. **Local maxima:** A local maxima is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards toward the peak, but will then be stuck with nowhere else to go. Figure 1 illustrates the problem schematically.
2. **Ridges:** A ridge is shown in Figure 2. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
3. **Plateaux:** A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which it is possible to make progress. (see Figure 1) A hill-climbing search might be unable to find its way off the plateau.

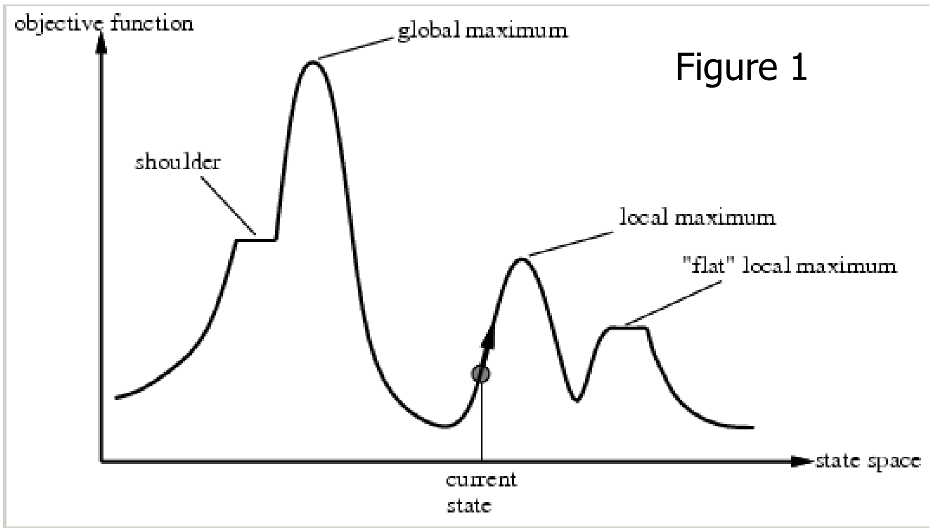


Figure 1

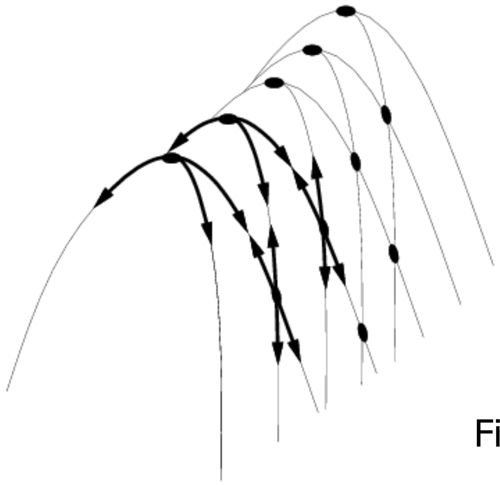


Figure 2

Hill Climbing on the 8 Queens Problem

```
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 ← the highest valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

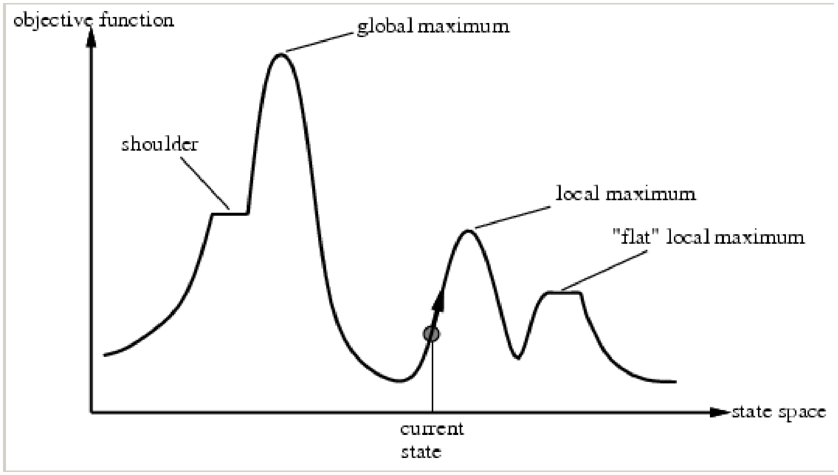
Local maxima | **Ridges** | **Plateaux**

In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with $8^8 \approx 17$ million states.

Sideways Move

Observation: The hill climbing algorithm halts if it reaches a plateau where the best successor has the same value as the current state.

Question: Might it not be a good idea to keep going—to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 1?



The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a **limit** on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill-climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

Stochastic Hill Climbing

Many variants of hill-climbing have been invented.

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

- **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g. thousands) of successors.

Random-Restart Hill Climbing

The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima.

- **Random-restart hill climbing** adopts the well known adage, "If at first you don't succeed, try, try again."
  - It conducts a series of hill-climbing searches from randomly generated initial states*, stopping when a goal is found. It is complete with probability approaching 1, for the trivial reason that it will eventually generate a goal state as the initial state.

*. Generating a *random* state from an implicitly specified state space can be a hard problem in itself.

Random-Restart Hill Climbing

If each hill-climbing search has a probability of p success, then the expected number of restarts required is $1/p$.

- For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success).
  - The expected number of steps is the cost of one successful iteration plus $(1-p)/p$ time the cost of failure, or roughly 22 steps.
  - When we allow sideways moves, $1/0.94 \approx 1.06$ iterations are needed on average and $(1\times21)+(0.06/0.94)\times64\approx25$ steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.*

*. Luby *et al*. (1993) prove that it is best, in some cases, to restart a randomized search algorithm after a particular, fixed amount of time and that this can be *much* more efficient than letting each search continue indefinitely. Disallowing or limiting the number of sideways moves is an example of this.

Random-Restart Hill Climbing

The success of hill climbing depends very much on the **shape** of the state-space landscape:

- if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly.
- On the other hand, many real problems have a landscape that looks more like a family of porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle, *ad infinitum*.

- NP-hard problems typically have an exponential number of local maxima to get stuck on. Despite this, a reasonably good local maximum can often be found after a small number of restarts.

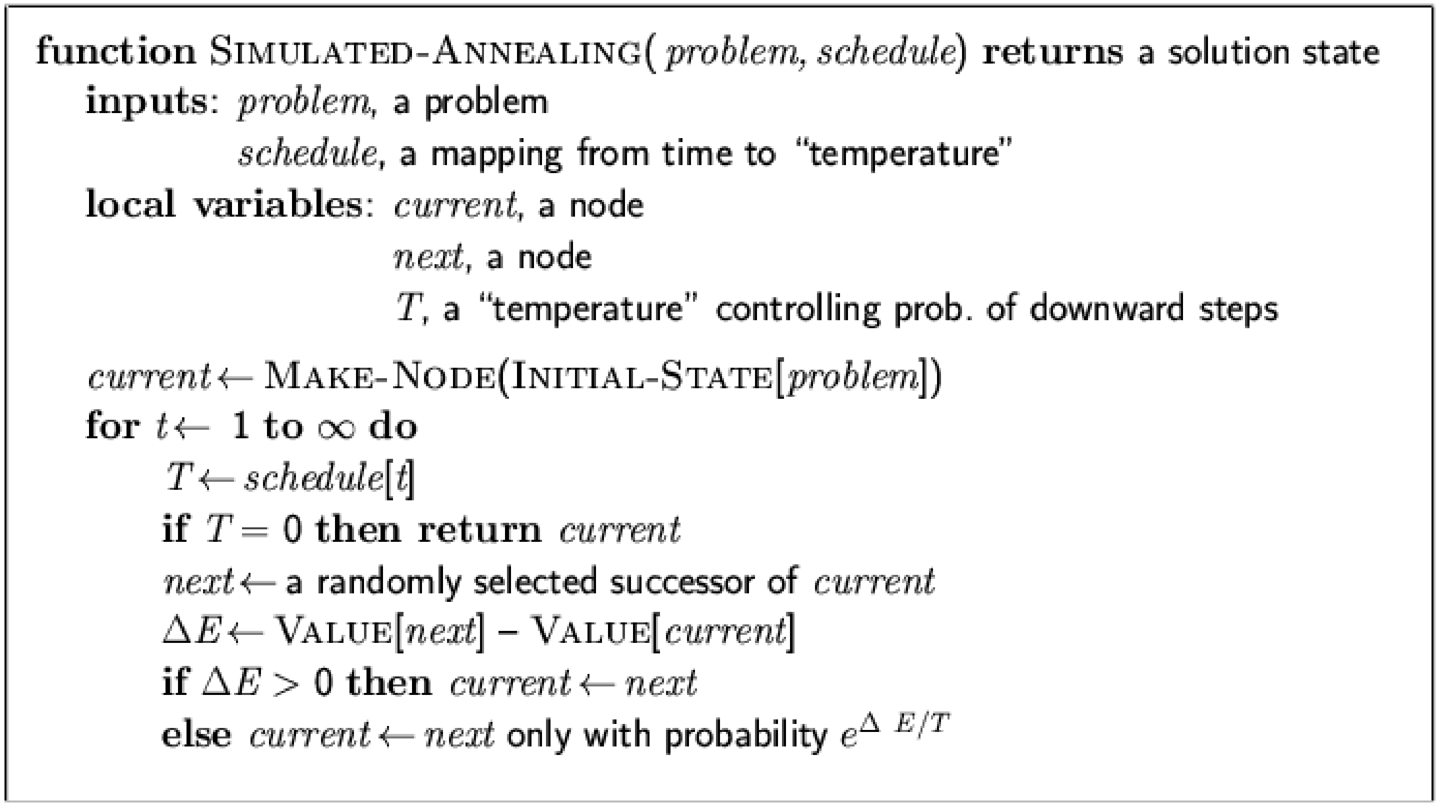
Simulated Annealing Search

A hill-climbing algorithm that never makes "downhill" moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.

- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete, but extremely inefficient.
  - Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness.
- **Simulated annealing** is such an algorithm.
- In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce into a low-energy crystalline state.

Simulated annealing search

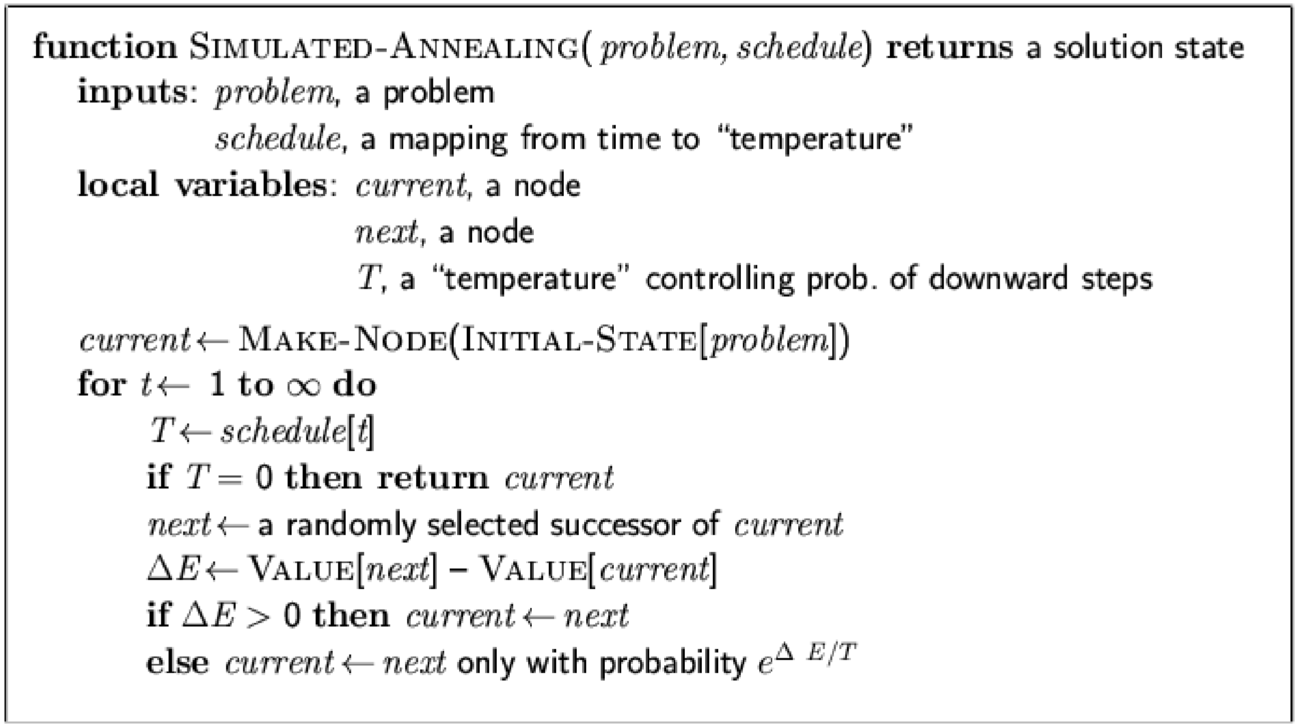
- *Idea: escape local maxima by allowing some "bad" moves, but gradually decrease their frequency*



```
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
        $\Delta E$ ← VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Properties of simulated annealing

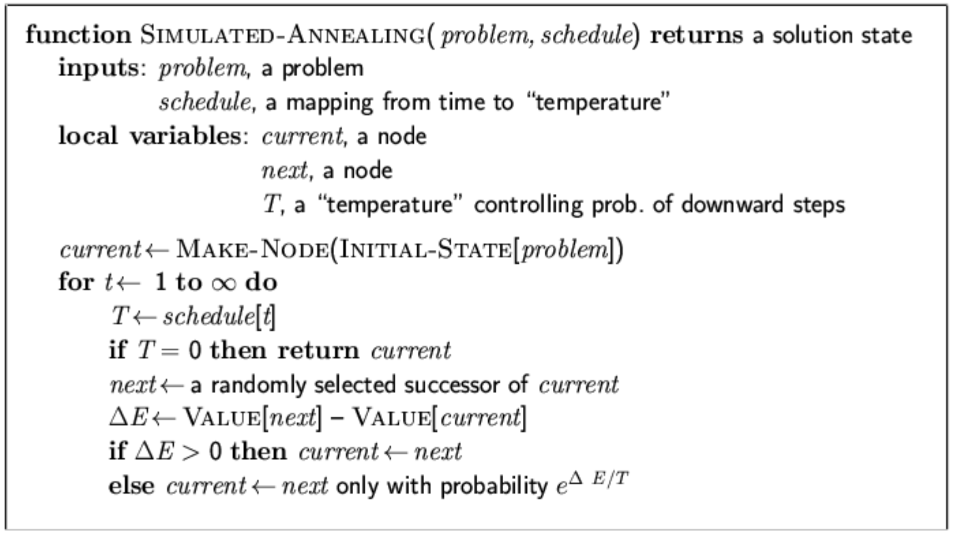
- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Widely used in VLSI layout, airline scheduling, etc



```
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
        $\Delta E$ ← VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Simulated Annealing

The innermost loop of the simulated-annealing algorithm is quite similar to hill climbing.



```
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 $e^{\Delta E/T}$
```

```
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 ← the highest valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then
            return STATE[current]
      current ← neighbor
```

– Instead of picking the *best* move, however, it picks a *random* move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when temperature is high, and they become more unlikely as T decreases. One can prove that if the *schedule* lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Local Beam Search

Keeping just one node in memory is an extreme reaction to the problem of memory limitations!!!!

Local Beam Search keeps track of *k* states instead of just one!!!

BEGIN: *k* randomly generated states

AT EACH STEP: generate all successors of all *k* states
select the best *k* successors
if any selected successor is GOAL, program ends

It seems to be the same as hill-climbing with k random restarts!

NO!!! Useful information is passed among all parallel search threads.

- Stochastic Beam Search

Genetic Algorithms

A variant of stochastic beam search in which each successor is generated by 2 parents!!!! The analogy to natural selection is the same as in beam search + sexual reproduction

BEGIN: *k* randomly generated states : called **POPULATION**

EACH STATE: an **INDIVIDUAL** – represented as a string over a finite alphabet.

Example: alphabet={0,1}

How do we represent INDIVIDUAL 7 ?? 111

How do we represent INDIVIDUAL 12 ??? 1100

REMEMBER the 8-QUEEN problem??? We need to specify the position of each of the 8 queens, each in a column, in each state. We will need $8 \times \log_2 8 = 24$ bits

Alternatively we could represent the state as 8 digits, each in the range 1-8.

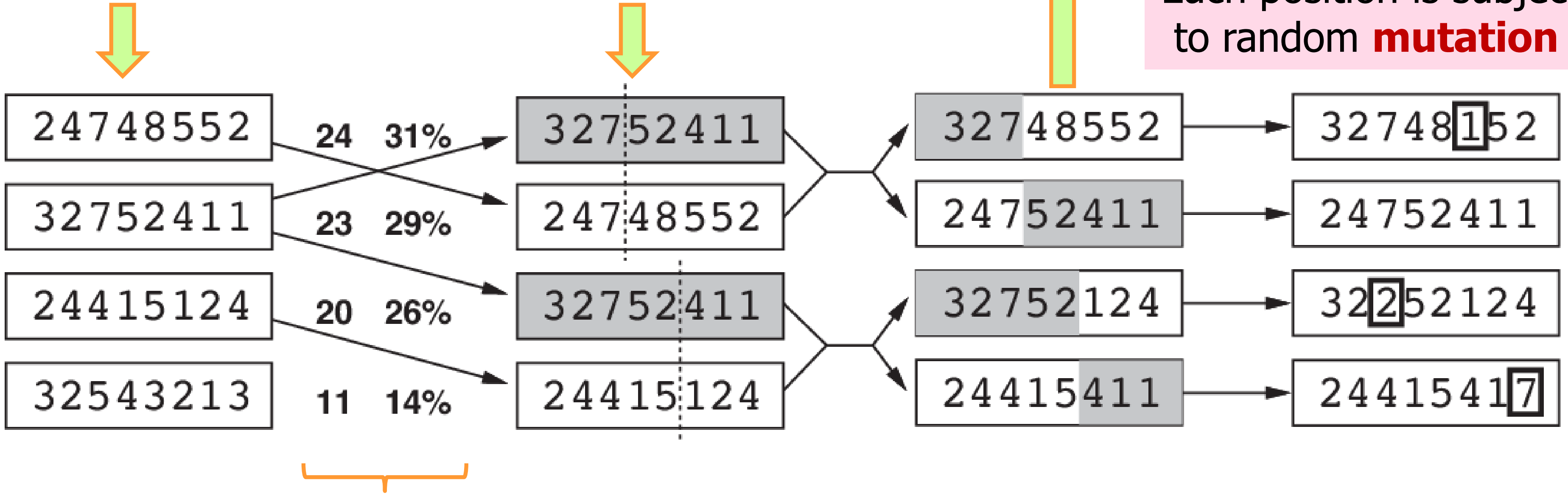
Representing the 8-Queen states

The **offsprings** are created by crossing over the parent strings at cross-over point

Each position is subject to random **mutation**

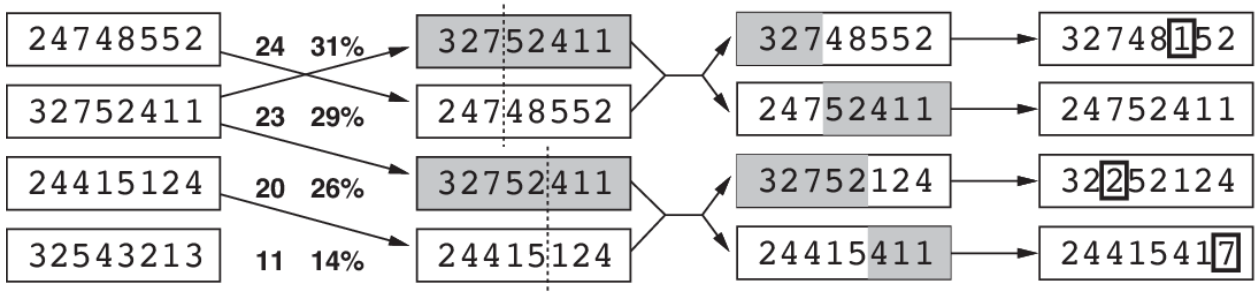
Initial **Population**

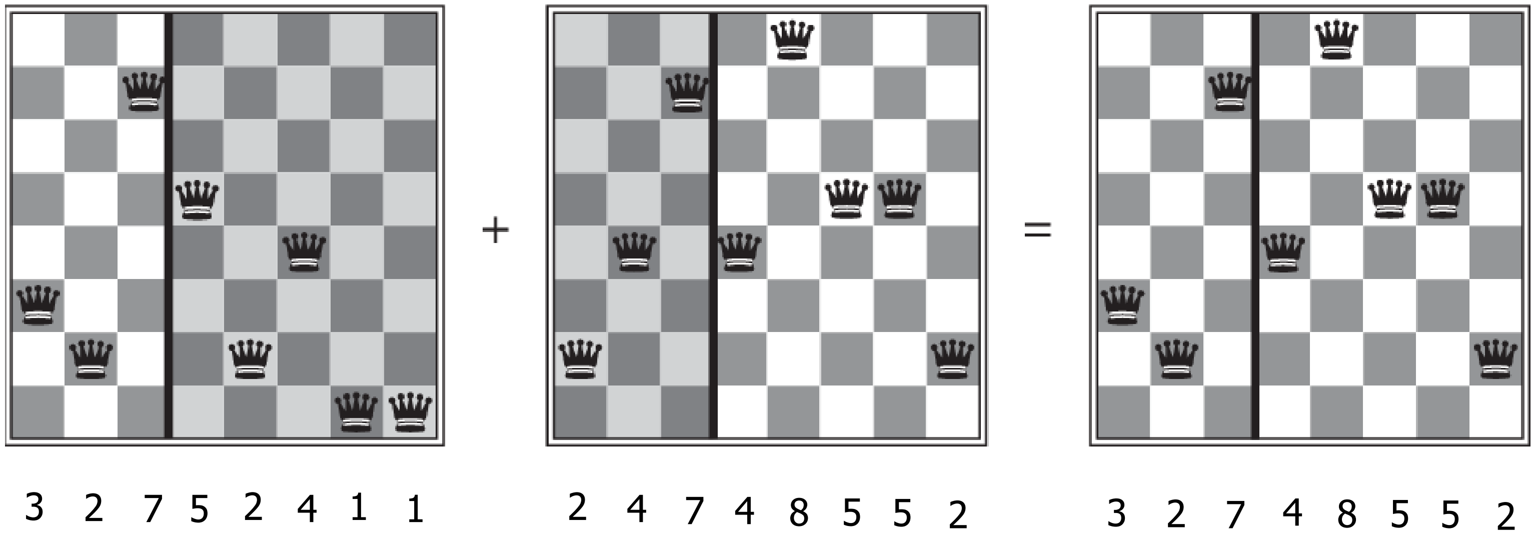
Selection (at random)



Fitness function: returns higher values for better states!!!
For the 8-Queen problem defined as # non-attacking pairs of queens
the probability of the state being chosen for reproduction is proportional to the fitness score

The 8-queen state offspring from 2 parents





A Genetic Algorithm

```
function Genetic-Algorithm (population, Fitness-Function) returns an individual
  inputs: population: a set of individuals;
          Fitness-Function : a function that measures the fitness of an individual

repeat
    new_population ← empty set
   for i=1 to Size(population) do
          x ← Random-Selection(population , Fitness-Function )
          y ← Random-Selection(population , Fitness-Function )
          child ← Reproduce(x , y )
  if (small random probability) then child ← Mutate (child )
until some individual is fit enough, or enough time has elapsed
return the best individual in population , according to Fitness-Function
```

```
function Reproduce (x , y )  returns an individual
  inputs: x , y : parent individuals

n ← Length(x ); c ← Random number from 1 to n
return Append(SubString(x , 1, c), SubString (y , c+1, n)
```

Online Search

- The agent first takes an action, then observed the environment, and computes the next action.
- Great for unknown environments!

The agent knows in each state *s* :

1. The actions allowed in *s: Actions(s)*
2. The step-cost function *c(s,a,s')*
3. **Goal-Test(s)**

But the agent does not know *Result(s,a)* unless they are in *s* doing *a*!

Read Section 4.5 in Textbook for examples of online search algorithms.