Local Search:
4. Path does not Matter

Outline

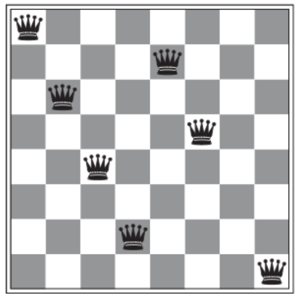
- Recap – informed search & heuristics
- Local search/Iterative Improvement algorithms/Optimisers
  - hill-climbing (gradient ascent/descent)
  - simulated annealing
  - local beams
  - genetic algorithms

Recap – Informed Search

- Informed search strategies may have access to heuristic functions $h(n)$ that estimate the cost of the shortest paths
- Greedy best-first search expands the lowest h
    - incomplete and not always optimal
- A* expands lowest $g + h$
    - complete and optimal (and optimally efficient)
- Admissible heuristics never overestimate the cost to reach the goal

The Story So Far...

- The search algorithms we've seen so far explore search spaces systematically by keeping one or more paths in memory and by keeping track of which alternatives have been explored at each point along the path
- When a goal is found, the path to the goal also constitutes a solution to the problem

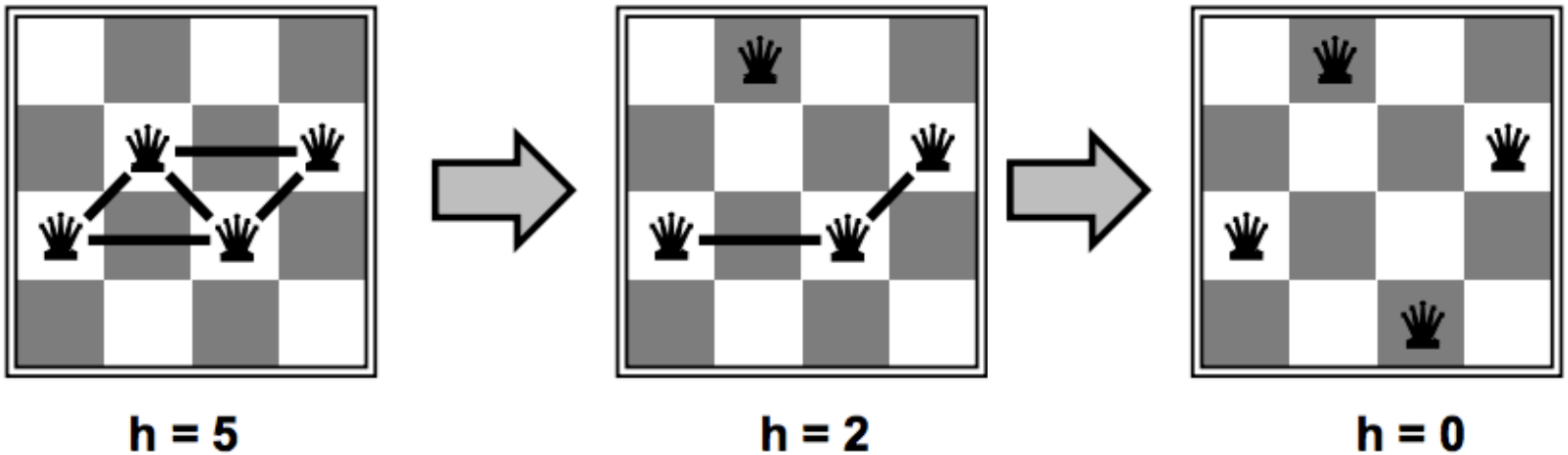


- But many problems do not care about the path, e.g. *8-queens*, just the final configuration and not which order the queens were added
- *Scheduling* problems (e.g. time allocation, resource allocation), just the final schedule matters, not how you arrived at that schedule
- In many *optimisation* problems, path is irrelevant, goal itself is the solution

Local Search

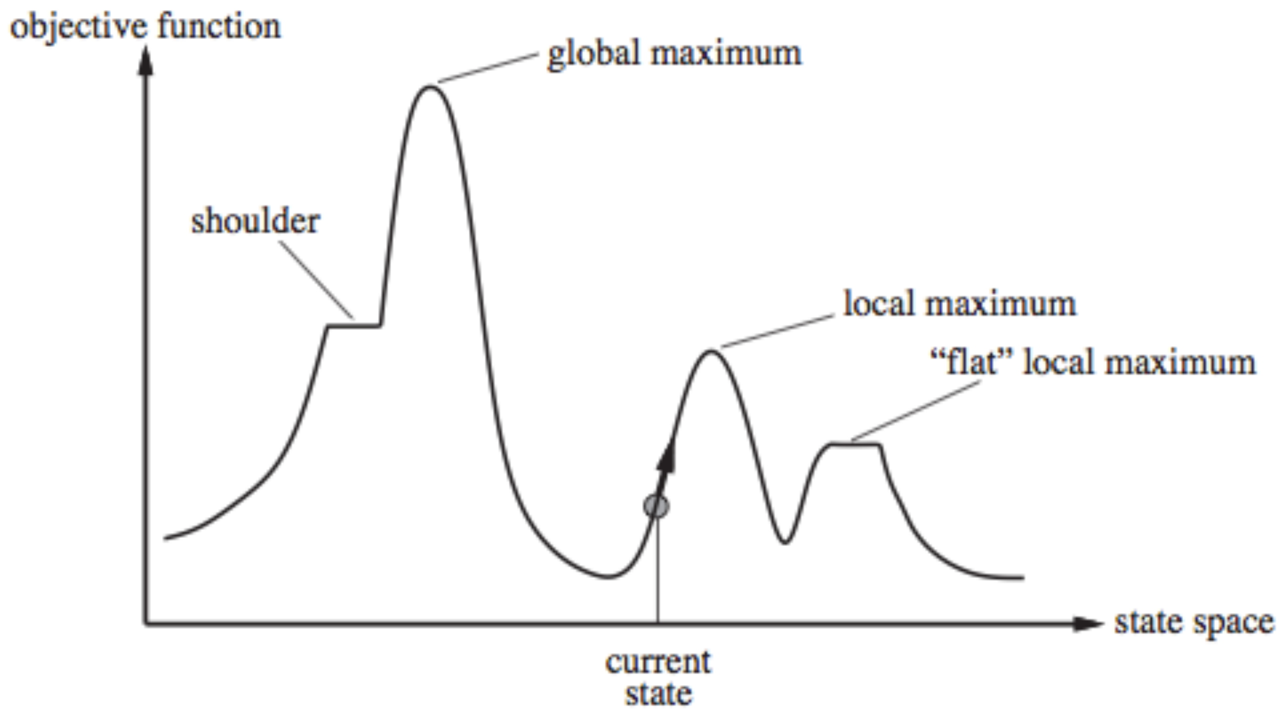
- Useable when solutions are states, not paths
- Start with a “complete” configuration (a single current state) and make modifications (iterative improvements) to improve its quality
- Given state space = set of complete configurations, find optimal configuration, (e.g. highest possible value or least cost) or find configuration satisfying constraints, (e.g. no two class at the same time in a timetable)
- Advantages of local search
  - they use very little memory (usually a constant amount)
  - can find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable
- Local search algorithms are very generic and have been applied successfully to many industrial problems

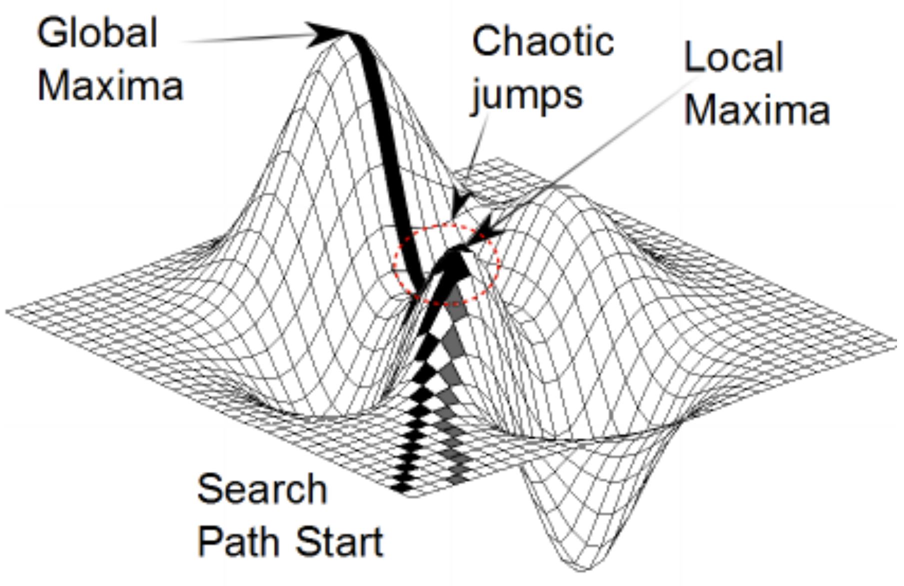
Example: n-Queens Problem



- Iterative improvement
  - start with one queen in each column (e.g. leftmost diagram)
  - move a queen along its column to reduce number of pairs of attacking queens (middle diagram)
  - repeat the previous step until *h* is minimum
- "We are what we repeatedly do." – Aristotle

State-space Landscape





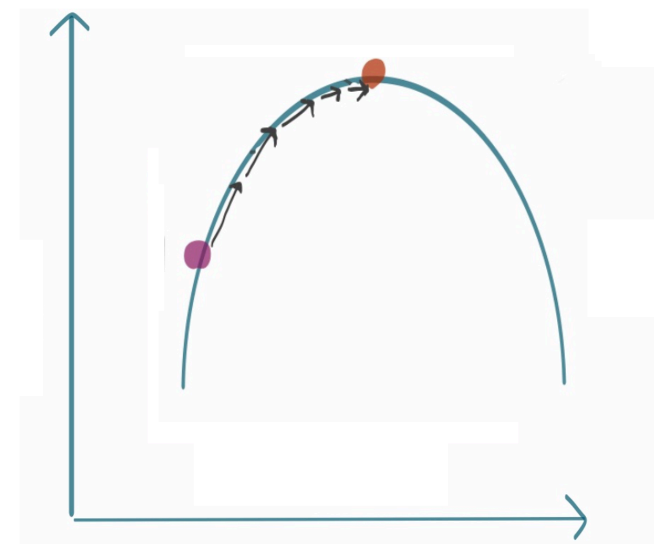
- A landscape has both a location (defined by the state) and an elevation (defined by the value for the heuristic cost function or objective function)
- If elevation corresponds to a cost, the aim is to find the lowest valley – global minimum
- If elevation corresponds to an objective function, the aim is to find the highest peak – global maximum
- Sometimes have to go sideways or even backwards in order to make progress towards the solution

Hill-climbing Search (Gradient Descent/Ascent)

- Most basic local search algorithm – sometimes called greedy local search
- A loop that continually moves in the direction of increasing value, i.e. uphill
- Terminates when it reaches a peak where no neighbour has a higher value
- Does not maintain a search tree, data structure for current node only records the state and the objective function
- Does not look ahead beyond the intermediate neighbours of the current state

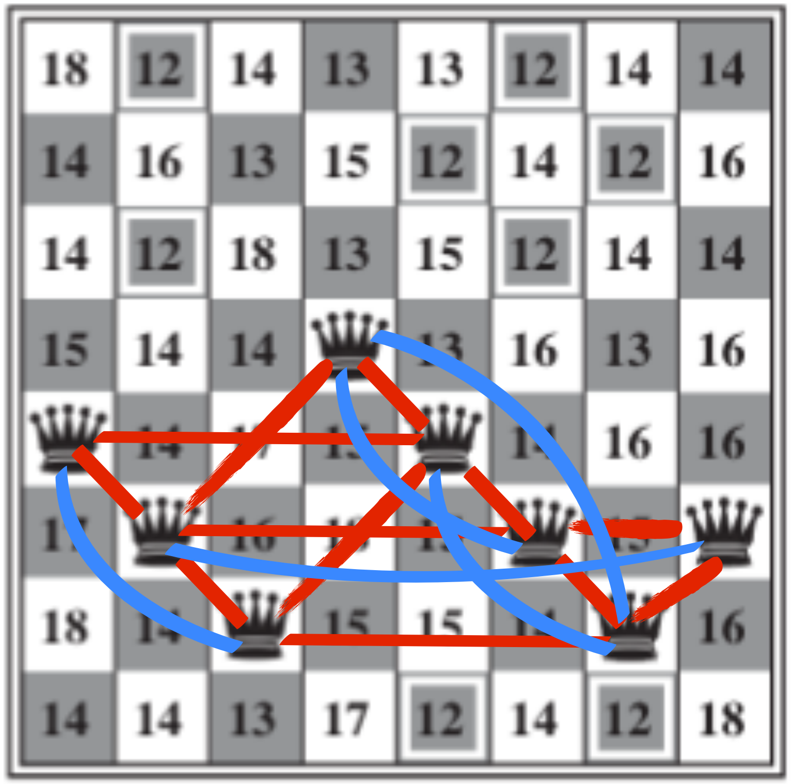
Hill-climbing Pseudocode

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   current ← MAKE-NODE(problem.INITIAL-STATE)
   loop do
      neighbor ← a highest-valued successor of current
      if neighbor.VALUE ≤ current.VALUE then return current.STATE
      current ← neighbor
```



- “Like climbing Everest in thick fog with amnesia”
- At each step the current node is replaced by the best neighbour; here it is the neighbour with the highest value
- If a heuristic cost estimate h is used, it would be the neighbour with the lowest h

8-queens using Hill-climbing



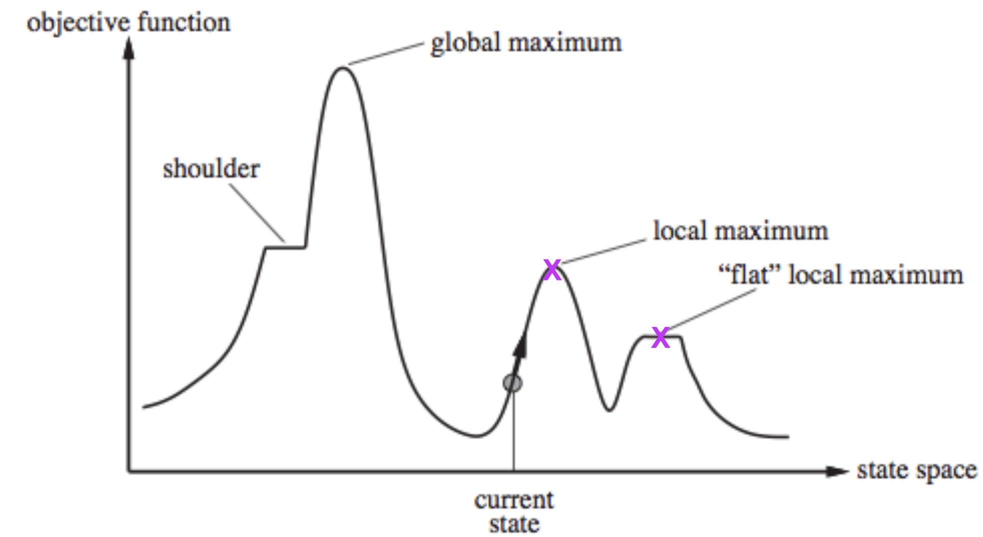
(a)

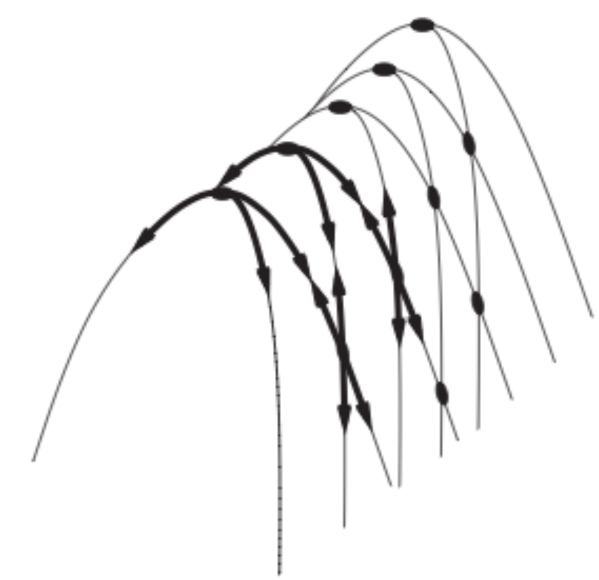
(b)

- Each state has 8 queens on the board, one per column
- Successors: all possible states generated by moving a single queen to square in the same column. Each state has 8 x 7 = 56 successor states (8 rows x 7 options per column)
- Heuristic cost function, *h* is the number of pairs of queens that are directly or indirectly attacking each other
- In (a), $h = 17$, the best moves from here are marked by the lowest numbers on the board by moving one queen within a column
- In 5 steps only, (b) can be reached
- In (b), $h = 1$, however, there are no further moves possible to decrease h, i.e. stuck at a local minimum (valley that is lower than is neighbouring states but higher than the global minimum)

Hill-climbing Problems

- Stuck at local optimum (maximum or minimum):
  - peak: higher than neighbours but lower than global maximum, no progress
  - ridge: oscillate from side to side
  - plateaux: a flat local maxima
- Potential solution: random restart
  - repeat the search from another starting point





Simulated Annealing

- Hill-climbing algorithm is incomplete – often fails to find a goal when one exists because they can get stuck at a local maximum or minimum
- Try combining hill-climbing with random walk – moving to a successor chosen at random from a set of successor states
- Idea: escape local maxima by allowing some “bad” moves (choose a worse neighbour) but gradually decrease their size and frequency
- Fun fact: the name annealing comes from the process used to harden metals and glass by heating them to a high temperature and then letting them cool slowly to reach low energy crystalline
- Start at a high intensity (e.g. high temperature) and then gradually reduce the intensity by lowering the temperature

Simulated Annealing Pseudocode* (Minimising)

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

T →∞: random walk (RW)
T →0: like hill-climbing (HC)

$$P(next) = \frac{1}{1 + e^{-\Delta E/T}}$$

**This algorithm was taken from the 4th edition of the textbook*

- In the innermost loop, instead of picking the *best* move (HC), it picks a random move
- Assessment: if the move improves the situation, it is always accepted, otherwise it is accepted with some probability less than 1 (RW)
- The probability decreases exponentially with the "badness" of the move, given by ΔE, the probability also decreases as the temperature T goes down

SA Intuition

- When T is high (beginning), probability of accepting “bad” moves is high (RW)
- When T is low (end), probability of accepting “bad” moves is low, only moves uphill (HC)

How is T Decreased?

- Some possibilities

    1. $T_n = T_{n-1}$ – constant
    2. $T_n = a\ T_{n-1}$ with $a < 1$
    3. T_n = constant / $(1 + n)$
    4. T_n = constant / $\ln(1 + n)$

When does SA Terminate?

- When the temperature becomes less than a given value
- When the number of iterations exceeds a given value
- When no improvement occurs after a given number of iterations

SA Example: Minimising Objective Function

- Stochastic hill-climbing, $f(current)=107$, $T=10$

$f(next)$	ΔE	$e^{-\Delta E/T}$	$P(next)$
80	27	0.067	0.94
100	7	0.497	0.67
107	0	1	0.5
120	-13	3.669	0.21

SA will accept this. Why?

$$P(next) = \frac{1}{1 + e^{-\Delta E/T}}$$

- The column $f(next)$ represents a set of candidate states that the current state can sample from at a time step
- Whether a randomly sampled candidate state will be accepted depends on ΔE and T

SA Example: Maximising Objective Function, f

- Stochastic hill-climbing, $f(current)=107$, $T=10$

$f(next)$	ΔE	$e^{\Delta E/T}$	$P(next)$
80	27	14.88	0.06
100	7	2.01	0.33
107	0	1	0.5
120	-13	0.27	0.78

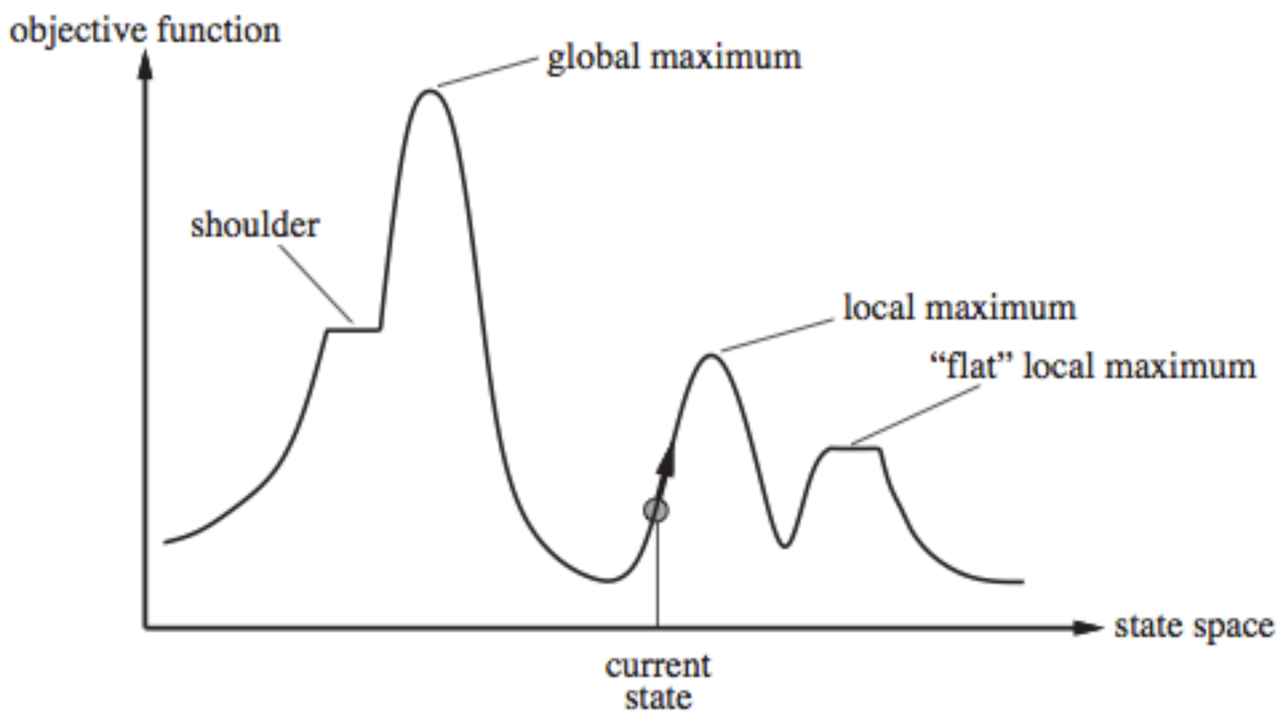
$\Delta E < 0$

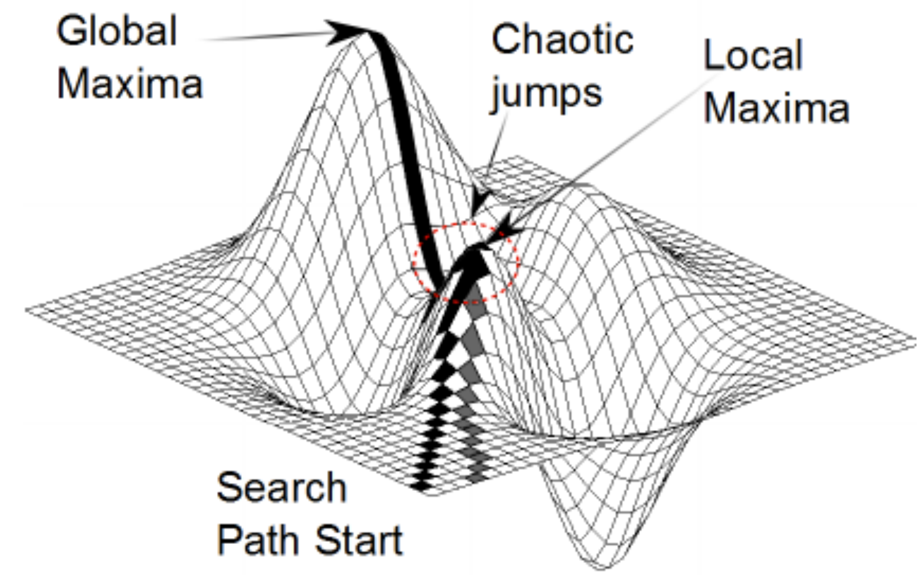
$$P(next) = \frac{1}{1 + e^{\Delta E/T}}$$

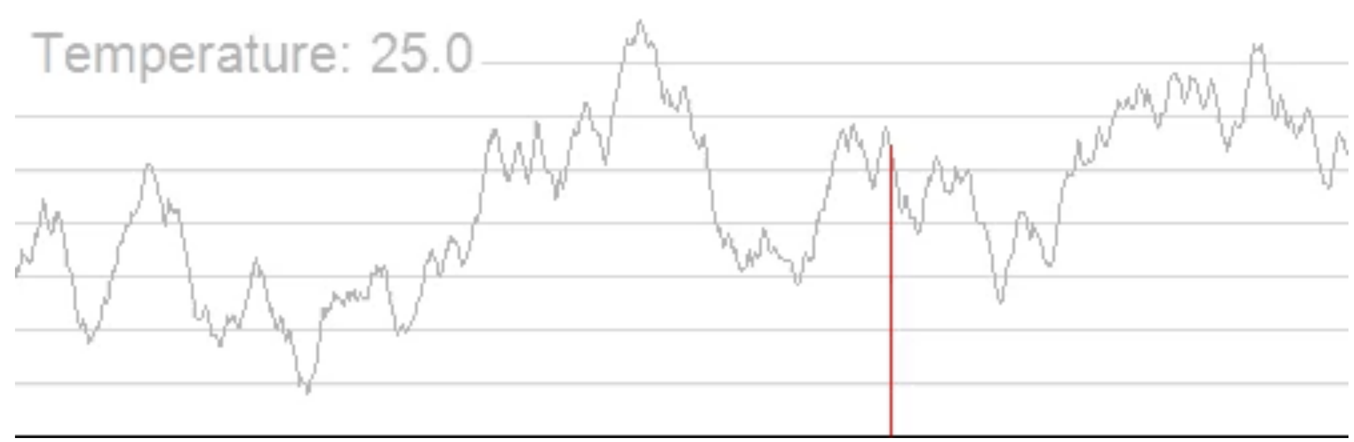
SA will accept this *Why?*

- For maximising, ΔE checking and probability calculation are slightly different
- $\Delta E = f(current) - f(next)$ should be negative for "better" states

Local Maxima/Minima







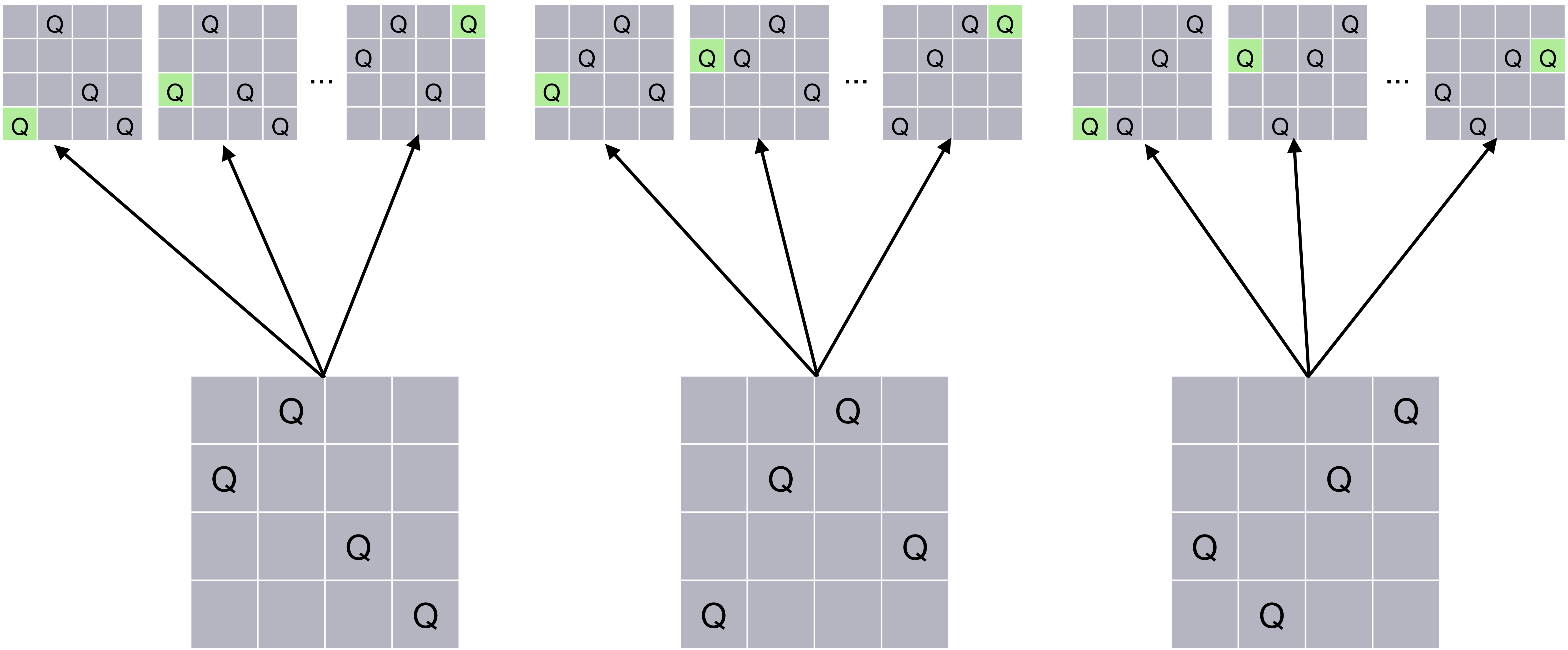
Local Beam Search

- Idea: Keep k states instead of 1; choose top k of all their successors
- Begin with k randomly generated states. At each step all the successors of *all k* states are generated. If any one is a goal, algorithm halts. Otherwise select k best successors from the list and repeat
- Problem: Quite often, all k states end up on same local hill (like HC)
- Stochastic beam search: Instead of choosing best k from successor candidates, choose k successors at random (like SA), biased towards good ones $\longrightarrow$ resembles natural selection; *successors* (offsprings) of a *state* (organism) populate the next generation according to its *value* (fitness)
- Special cases:
  - When $k = 1$, equivalent to HC
  - When $k = \infty$, equivalent to BFS

4-Queens Local Beam Search

$$k = 3$$

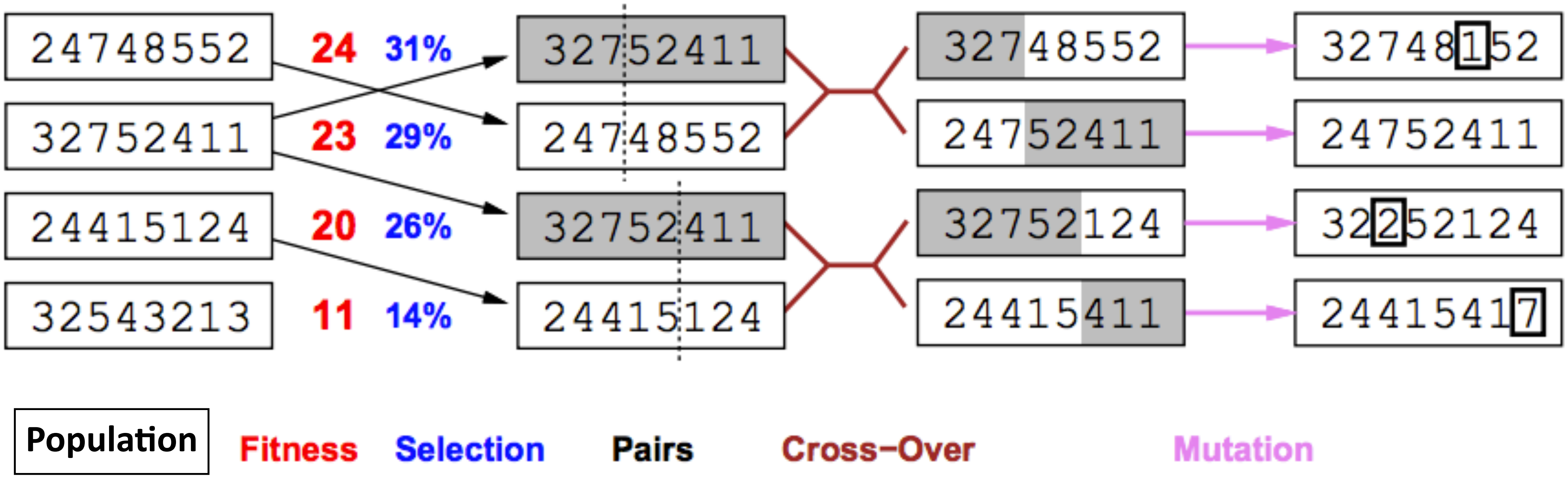
Until goal is found, choose "best" 3 states from all these successor states and repeat



Genetic Algorithms

- A variant of stochastic beam search in which successor states are generated by combining two parent states instead of one state
- Like beam searches, begin with a set of k randomly generated states, called the population
- Each state, or individual is represented as a string over a finite alphabet, most commonly 0s and 1s and is rated by the fitness function (objective function) – should return higher values for better states
- Based on this, pairs of individuals are selected and mated by choosing a crossover point from positions in the string
- Offsprings are created by crossing over the parent strings at the crossover point. Finally each location is subject to random mutation (e.g. in 8-queens, choosing a queen at random and moving it to a random square in its column)

GA Illustrated (8-queens)

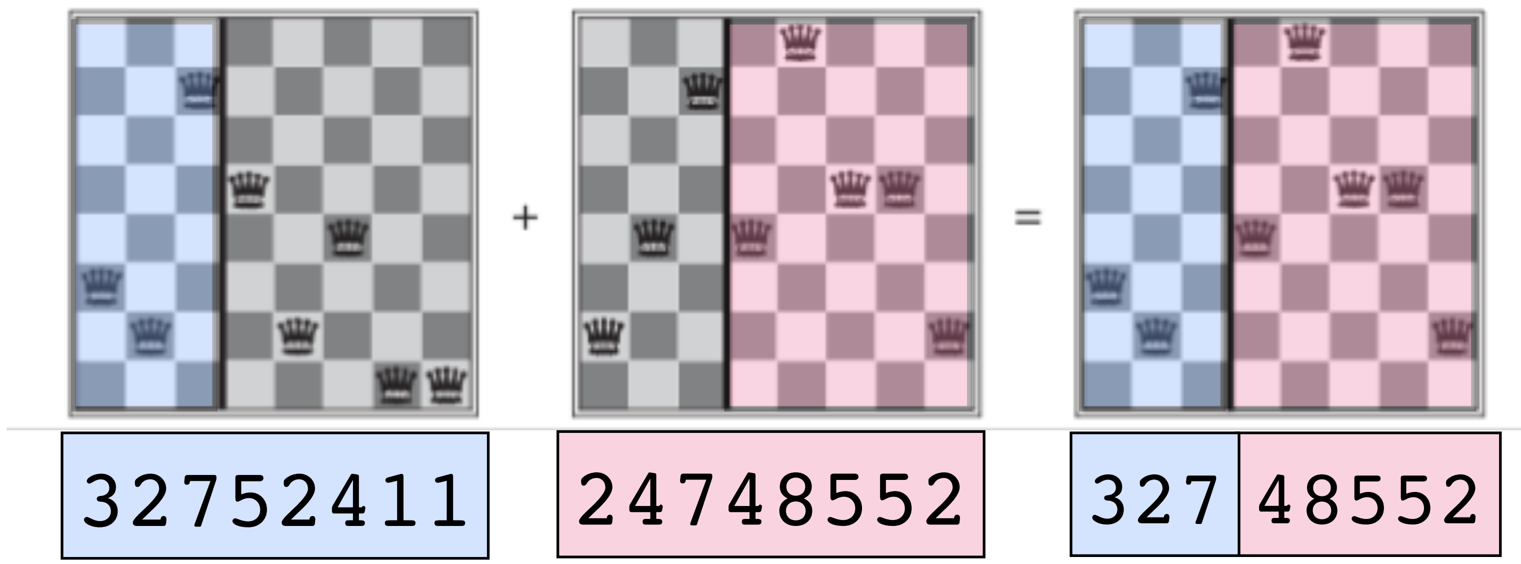


- Population consists of individuals that are candidate solutions. Digits represent positions of queens on each column. Think of them as chromosomes.
- **Fitness** function: the number of *nonattacking* pairs of queens (28 for solution) = objective function
- Probability of being **selected** for reproduction (%) is directly proportional to the fitness score (= exploitation)
- **Pairs** – The individuals chosen are paired off at random; one of them has been chosen twice while one hasn't been chosen at all
- **Crossover** – when two parent states are quite different, produced states can be a long way from either parent, so crossover (like SA) frequently takes large steps in the state space early and smaller steps later on when most individuals are quite similar (= exploitation)
- **Mutation** – each location is subject to random mutation. In 8-queens, moving any queen to another position in the same column (= exploration)

Exploitation vs Exploration

- Exploitation – using already exist solutions and make refinement to it so its fitness will improve
  - Selection: use of solutions with high fitness to pass on to next generations
  - Crossover: main role is to provide mixing of the solutions and convergence in a subspace
- Exploration – the algorithm searches for new solutions in new regions
  - Mutation: the change of parts of one solution randomly, which increases the diversity of the population and provides a mechanism for escaping from a local optimum. Leads to a solution outside the subspace

GA: Pseudocode



✱ Question
What would a mutation in the 6th digit (to the string 32748152) result in?

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

A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

Only produces one offspring, not two, but k overall

Termination:
- $f(n)$ reached predefined value
- Cutoff time/generations
- No improvement in $f(n)$

Perform crossover between a pair and return one string/child

GA: Worked Example

- Problem: maximise the function $f(x) = x^2$ where x is is between 1 and 31
- Variables codes as strings. Numbers between 1 and 31 can be coded as 5-digit binary string, 00001_2 (1_{10}), 01010_2 (10_{10}), 11111_2 (31_{10})
- Start with population size 4, randomly
- Fitness/Objective function is x^2

Population: Iteration 0, Generation 0

Initial Population	x	$f(n) = x^2$
01101	13	169
11000	24	576
01000	8	64
10011	19	361

- Sum of `f(n)`, `Σf` = 169 + 576 + 64 + 361 = 1170
- Average of `f(x)` = 293
- `max(f(x))` = 576

Selection

$f(x) = x^2$	$f(x)/\Sigma f$	f(x)/	x
169	0.14	0.58	13
576	0.49	1.97	24
64	0.06	0.22	8
361	0.31	1.23	19

- Many selection algorithms, e.g. Roulette wheel
- The best strings get more copies, weak ones die off
- After selection, crossover takes place
- Apply one bit mutation

Crossover & Mutation

Selected	String	Crossover	Mutation
13	01101	01100	01101
24	11000	11001	11101
19	10011	10000	10100
24	11000	11011	11001

- At which point was crossover for 1st pair and 2nd pair?
- Which bit was mutated in each child?

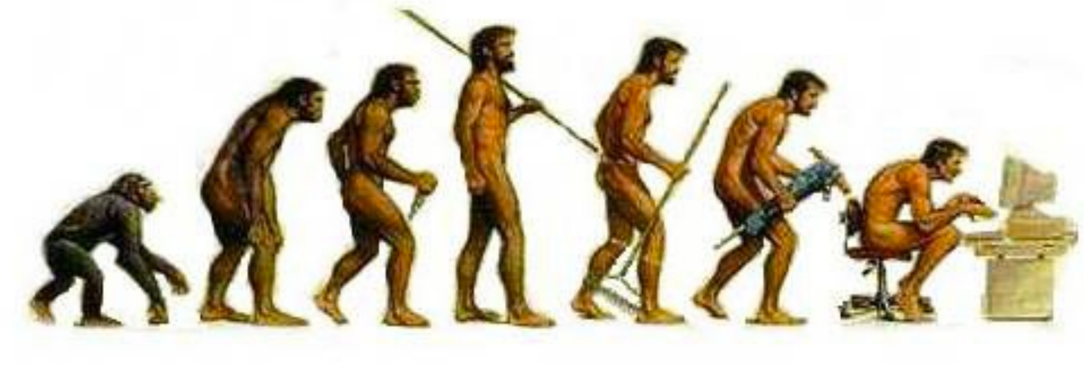
Next Generation, G1

New Population	x	$f(x) = x^2$	$f(x)/\Sigma f$
01101	13	169	0.08
11101	29	841	0.41
10100	20	400	0.2
11001	25	625	0.31

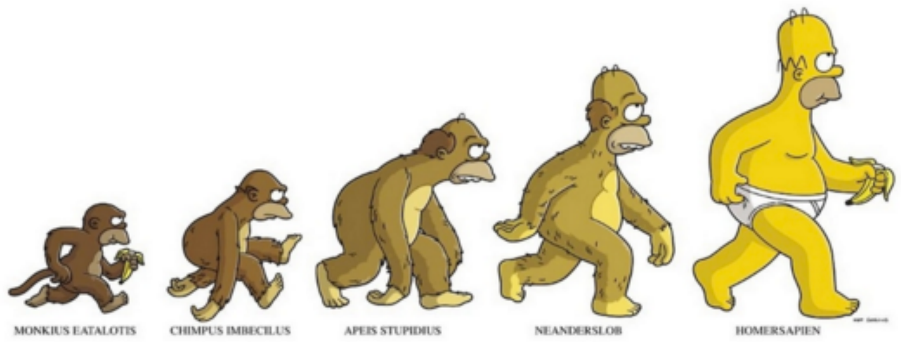
- `Σf` = 169 + 841 + 400 + 625 = 2035
- Average of `f(x)` = 508
- `max(f(x))` = 841
- Is this population better than initial population?

Initial Population:
`Σf` = 1170
`mean(f(x))` = 293
`max(f(x))` = 576

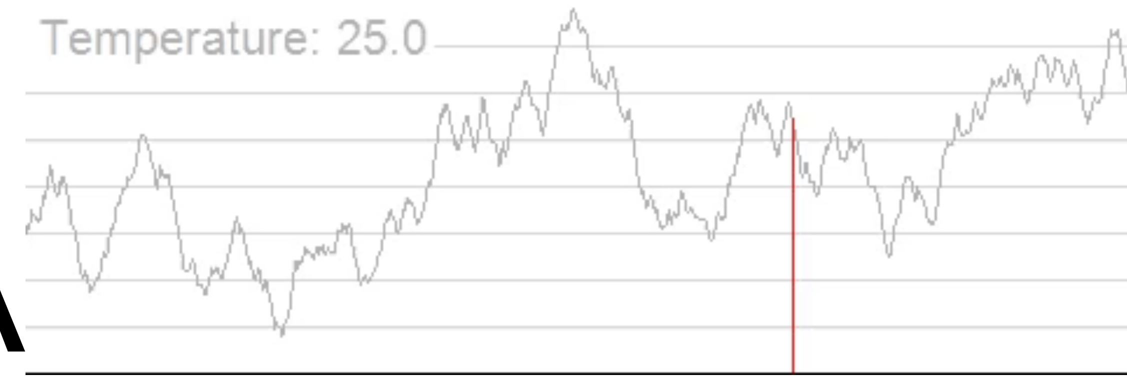
GA – Discussion



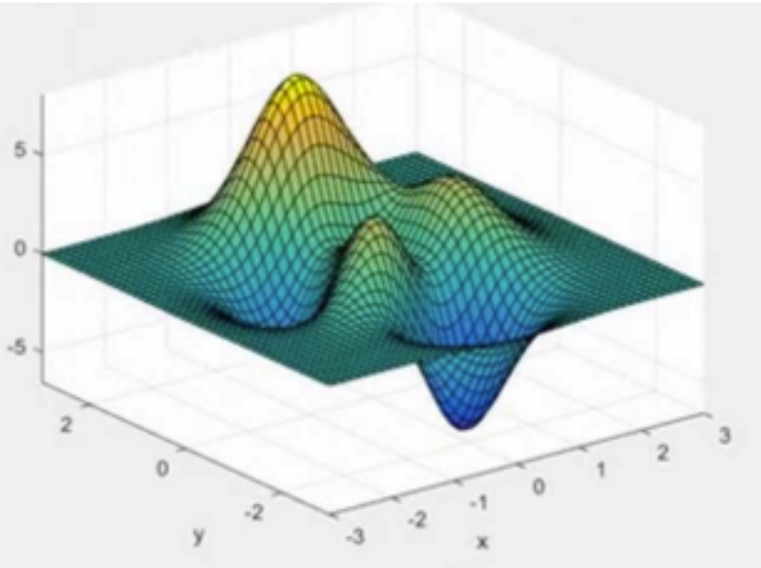
- Work well for continuous and discrete problems
- Tend not to get stuck in local maxima/minir (thanks to mutation)
- Computationally expensive
- Easy to perform in parallel
- Objective (fitness) function may be hard
- Practical applications: Knapsack problem, feature selection, travelling salesman (TSP)



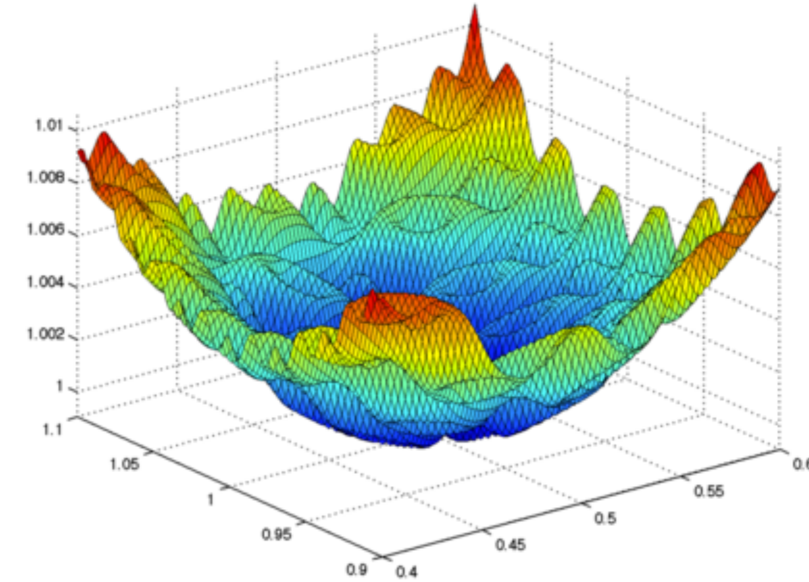
GA vs SA



- Both are nature-inspired
- Both can be used to solve same hard optimisation problems
- SA is a single state method, GA is a population method
- GA is normally better than SA performance-wise but at higher computation cost (can be distributed)
- SA might outperform GA in problems where solution space is small



Summary



- Hill-climbing: greedy local search that continually moves uphill; incomplete as can get stuck in local maxima
- Simulated annealing (non-deterministic/stochastic/randomised)
  - explore successors wildly randomly (High Temp)
  - as time goes by, explore less wildly (Cool down)
  - until there is a time where things settle (Cold)
- Local beam: keep k states instead of 1
- Genetic algorithms: local beam search + generate successor(s) from pairs of states – "survival of the fittest"
- SA and GA have been used in many optimisation problems – VLSI layout problems, factory scheduling, airline scheduling and other large-scale optimisation tasks

References

- Russel and Norvig, Chapter 4
- Paper on Simulated Annealing: Metropolis *et al.* (1953). "Equation of State Calculations by Fast Computing Machines", Journal of Chemical Physics. 21(6)
- Genetic algorithms explanation (Georgia Tech) [Link]
- The Knapsack Problem & Genetic Algorithms (Computerphile) [Video]
- GA demo – Learning to walk [Link]
- GA demo – Helping snakes find their way to food (G. Muric) [Video] [Code]