

Local Search

Simon Dixon
Queen Mary University of London

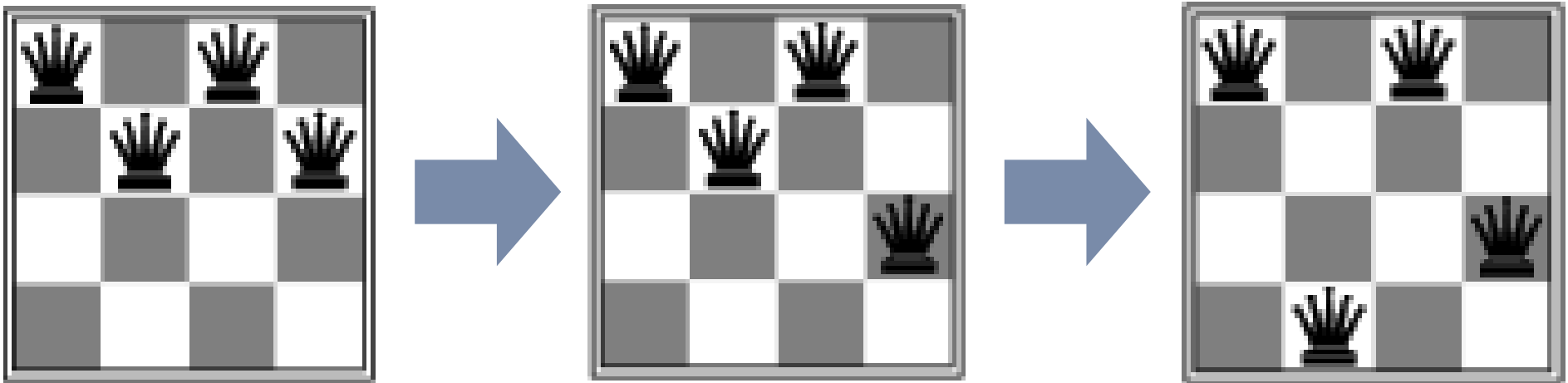
- Learn about
 - ▶ Local search algorithms
 - ▶ Hill-climbing search
 - ▶ Simulated annealing search
 - ▶ Local beam search
 - ▶ Genetic algorithms

- Good for optimisation problems
 - ▶ no goal test
 - ▶ the *path* to the goal is irrelevant
 - ▶ the goal *state* itself is the solution
 - integrated circuit design
 - factory-floor layout
 - automatic programming
 - telecommunications network optimisation
 - ▶ the performance measure distinguishes between alternative solutions
- Find configurations satisfying constraints
 - ▶ e.g. n-queens (see slide 5)

- State
 - ▶ keep a single "current" state, try to improve it
 - ▶ cf. searching a tree of possible paths for the optimal sequence
- Advantages
 - ▶ Use little memory (usually constant amount)
 - ▶ Can find reasonable solutions in large or infinite (continuous) problem spaces

Example: the n-Queens puzzle

- Put n queens on an $n \times n$ chess board such that no queen can attack another
 - ▶ a queen can attack any piece on the same row, column, or diagonal as itself



Failed “solutions” to the 4 Queens problem

- A state has
 - ▶ Location
 - given by the values of the state variables
 - ▶ Elevation
 - its heuristic cost or objective value
 - note: we can't see the whole function; we can calculate it for individual points
- Solutions (two ways of thinking about it, but equivalent)
 - ▶ Heuristic cost (find minimum cost) : global minimum.
 - ▶ Objective function (find best state or maximal utility): global maximum

State space landscape

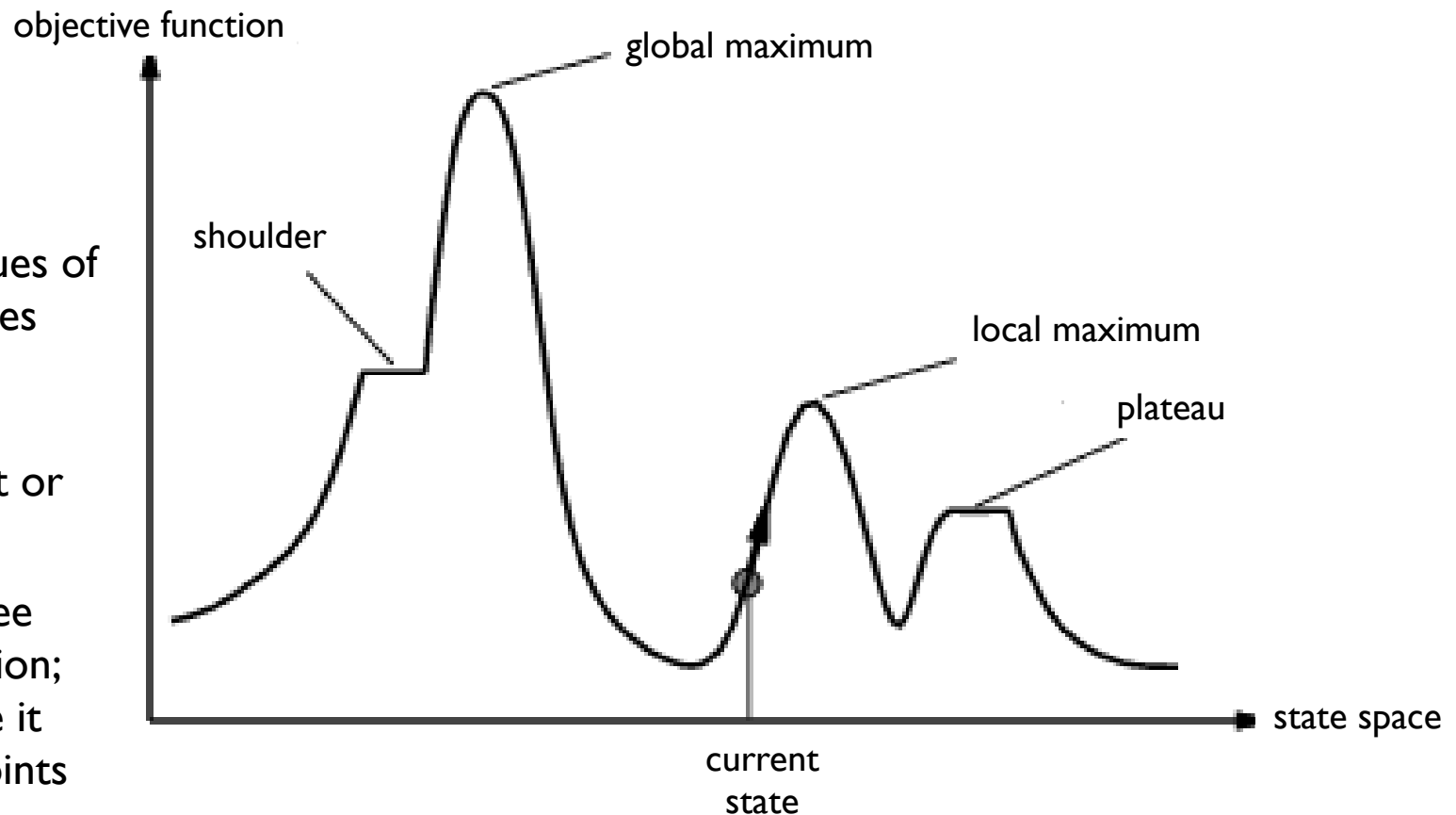
- A state has

- ▶ Location

- given by the values of the state variables

- ▶ Elevation

- its heuristic cost or objective value
 - note: we can't see the whole function; we can calculate it for individual points



- Solutions (two ways of thinking about it, but equivalent)

- ▶ Heuristic cost (find minimum cost) : global minimum.

- ▶ Objective function (find best state or maximal utility): global maximum

- “Local” because we don’t care about the path through the search space
- A local search algorithm is
 - ▶ *complete*: if it always finds a goal if one exists
 - ▶ *optimal*: if it always finds the global minimum/maximum
- Problem: depending on the initial state, search can get stuck in local maxima or minima

Hill-climbing search









- Like agenda-based search, but the agenda contains only one item
 - ▶ so we take the best option at each stage, and throw away all the others

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

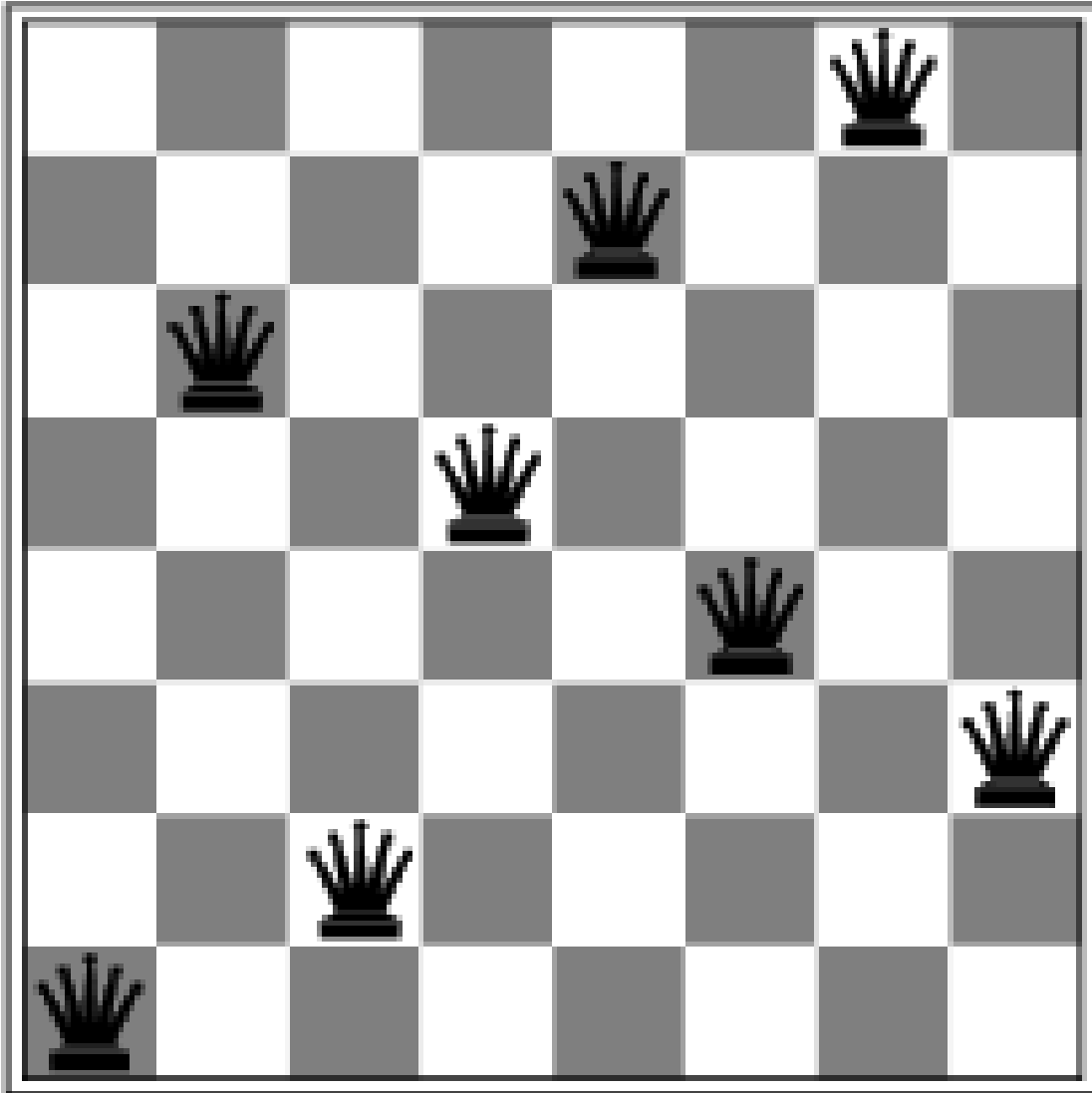
  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```

Hill-climbing: the 8-queens problem

- Local search: use complete states
 - ▶ always 8 queens on board
 - ▶ one queen per column
- Successors
 - ▶ set of states generated by moving one queen *vertically*
- Heuristic
 - ▶ h = number of pairs of queens attacking each other, directly or indirectly
 - ▶ $h = 17$ for the example
 - ▶ $h = 0$ is a solution state (global 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

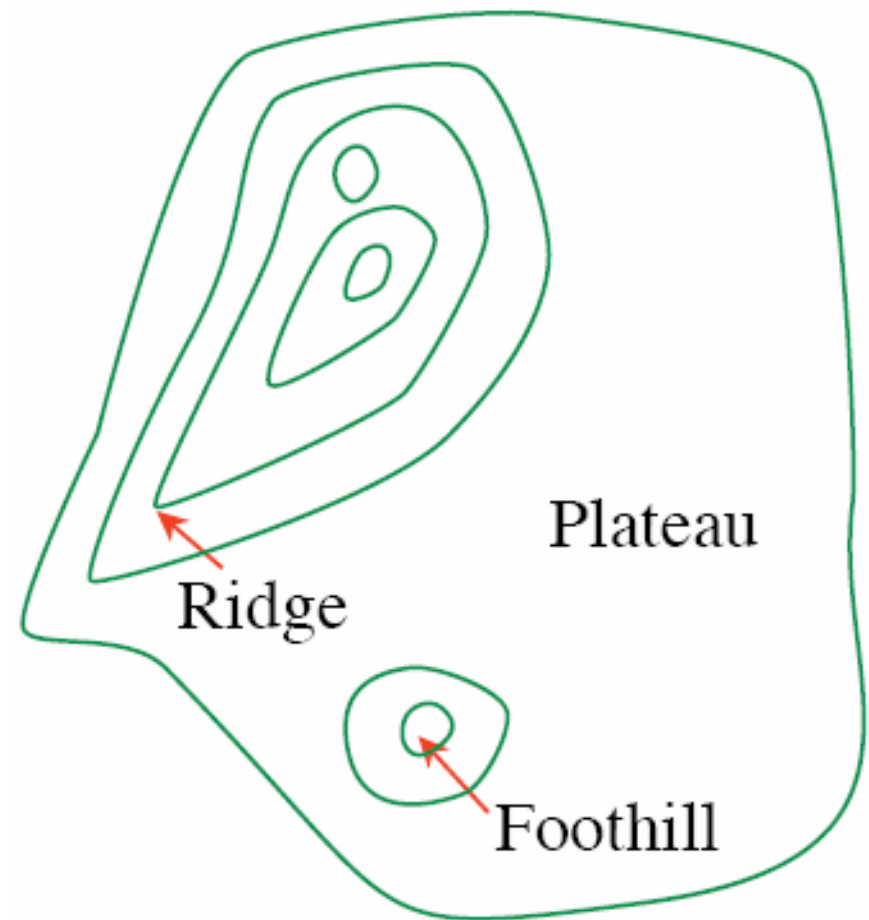
Hill-climbing: the 8-queens problem



- A local minimum with $h = 1$
- What is the set of successor states?
- Why is this a local minimum?
- What is the next step of hill-climbing?

Hill-climbing search

- Greedy local search
 - grabs best neighbour without “thinking ahead”
- Rapid improvement in heuristic value
 - But at a fatal cost
 - Local Maxima/Minima will stop search altogether
 - Ridges: a sequence of local maxima that make it difficult to escape.
 - Plateaux: heuristic values uninformative
 - Foothills: local maxima that are not global maxima



How bad is the problem?

- Algorithm reaches a point where no progress is made
- To test on the 8-queens example:
 - ▶ run many searches starting from random configurations of 8-queens
 - too many to do it exhaustively
 - ▶ result
 - 86% of starting configurations get stuck
 - 14% of starting configurations succeed
 - ▶ the algorithm works quickly, usually only evaluating 3-4 steps before either getting stuck or finding a solution

How can we solve the problem?

- Allow different moves?
 - ▶ e.g., sideways
 - ▶ must account for local maxima (foothills) otherwise could get stuck in an infinite loop
 - e.g., cap number of sideways moves
 - but we risk losing generality with such ad hoc solutions
- Result:
 - ▶ 6% stuck
 - ▶ 94% solution
- Works much slower
 - ▶ evaluating 21-64 steps before either getting stuck or finding a solution

- Stochastic hill-climbing
 - ▶ choose randomly between available uphill moves
 - choose from uniform or non-uniform distribution
 - ▶ converges more slowly but sometimes finds better solutions
 - optimality still not guaranteed in all cases
- First-choice hill-climbing
 - ▶ Good for big problems
 - Uses Stochastic hill climbing but randomly generates successors and picks first larger one
- Random-restart hill-climbing
 - ▶ If at first you don't succeed, try again, starting from a different place
 - requires that non-optimality can be recognised

- Hill-climbing
 - ▶ only improves on the current solution
 - ▶ not complete
 - may get stuck in local minima/maxima
- Random walk
 - ▶ moves from state to state randomly
 - ▶ complete (given infinite time) but very inefficient
- Simulated Annealing combines these two to give a compromise between search complexity and completeness

- Main idea: Escape local maxima by allowing some bad moves
- Gradually decrease frequency of allowed bad moves as search proceeds
- By analogy with a process of hardening in steel production
 - ▶ want crystal structure of metal to be all lined up
 - ▶ heat the metal to make the crystals vibrate and “shake out” irregularities
 - ▶ let it cool slowly so that the more ordered form stays ordered

Simulated Annealing search

- So in the algorithm, we have an imaginary notion of “temperature”, which allows random movement outside the hill-climb
 - as the “temperature” drops, less random movement is allowed

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

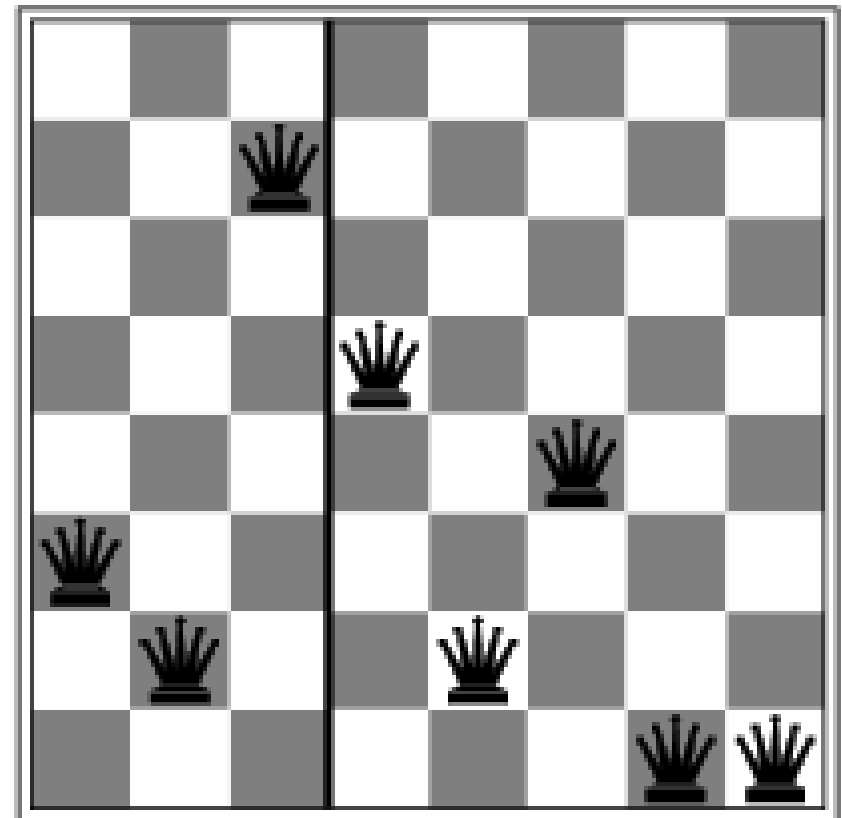
- Like getting a ping-pong ball into the deepest crevice of a bumpy surface
 - ▶ Left alone by itself, ball will roll into a local minimum
 - ▶ If we shake the surface, we can bounce the ball out of a local minimum
 - ▶ The trick is to shake hard enough to get it out of local minimum, but not hard enough to dislodge it from global one
 - ▶ We start by shaking hard and then gradually reduce the intensity of shaking
- If the “temperature” decreases slowly enough, simulated annealing search finds a global optimum with probability approaching 1.0
- Widely used for optimisation problems such as VLSI layout, airline scheduling, etc.

- Main idea: Keep track of k states rather than just one
- Start with k randomly generated states
- At each iteration
 - ▶ generate all successors of all k states
 - NB: not the same as an agenda k long!
 - ▶ if any one is a goal state, stop
 - ▶ else select the k best successors from the complete list and repeat
- So this is like a kind of selective breadth first search

- Local beam search looks like running k hill-climbing algorithms in parallel, **but it is not**
 - ▶ the results of all k states influence each other
 - ▶ if one state generates several good successors, they all end up in the next iteration
 - ▶ states generating bad successors are weeded out
- This is both a strength and a weakness:
 - ▶ unfruitful searches are quickly abandoned and searches making the most progress are intensified
 - ▶ can lead to a lack of diversity: concentration in a small region of the search space
 - ▶ remedy: choose k successors randomly, biasing choice towards good ones; or explicitly avoid keeping multiple similar solutions

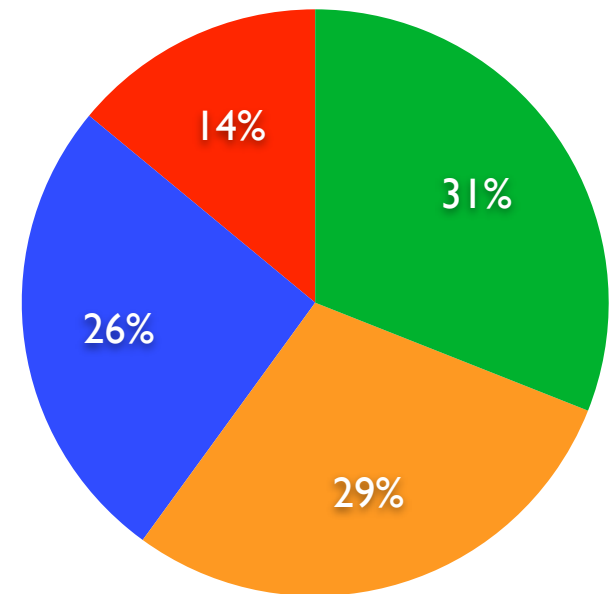
- Genetic Algorithms (GAs) are a variant of local beam search
- A successor state is generated by combining two parent states
- Start with a large number of randomly generated states (a *population*)
- A state is represented as a binary string
- An *evaluation* function or *fitness* function assesses the quality of a state
 - ▶ higher values for better states
- Produce the next generation of states by
 - ▶ selection
 - ▶ crossover
 - ▶ mutation

- The way that you encode the problem influences the search
- Example: 8 queens, 8x8 board
 - ▶ 64 bits, one for each square? (= 8x8 matrix)
 - ▶ 8 octal digits? (one 3-bit row number for each column)
 - 21641300 (octal)
 - 010001110100001011000000 (binary)
- Fitness function: no. of non-attacking pairs of queens (min = 0, max = 28)
 - ▶ Fitness here =
 $6 + 5 + 4 + 3 + 3 + 2 + 0 = 23$



- After encoding states and calculating fitness, select pairs for reproduction
 - ▶ many methods can be used
- Simplest: randomly choose pairs of states with non-uniform probability

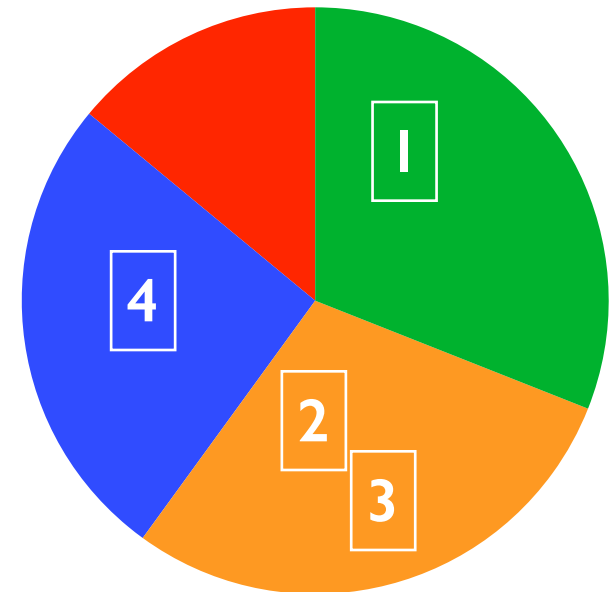
State	Fitness	P(selection)
13637441	24	$24/(24+23+20+11) = .31$
21641300	23	$23/(24 + 23 + 20 + 11) = .29$
13305013	20	$20/(24 + 23 + 20 + 11) = .26$
21432102	11	$11/(24 + 23 + 20 + 11) = .14$



GA: Selection for Reproduction

1	136	37441
2	216	41300
3	21641	300
4	13305	013

State	Fitness	P(selection)
	24	$24/(24+23+20+11) = .31$
	23	$23/(24 + 23 + 20 + 11) = .29$
	20	$20/(24 + 23 + 20 + 11) = .26$
21432102	11	$11/(24 + 23 + 20 + 11) = .14$



GA: Selection for Reproduction

1	136	41300
2	216	37441
3	21641	013
4	13305	300

Cross-over leads to new population

GA: Selection for Reproduction

1 13641300

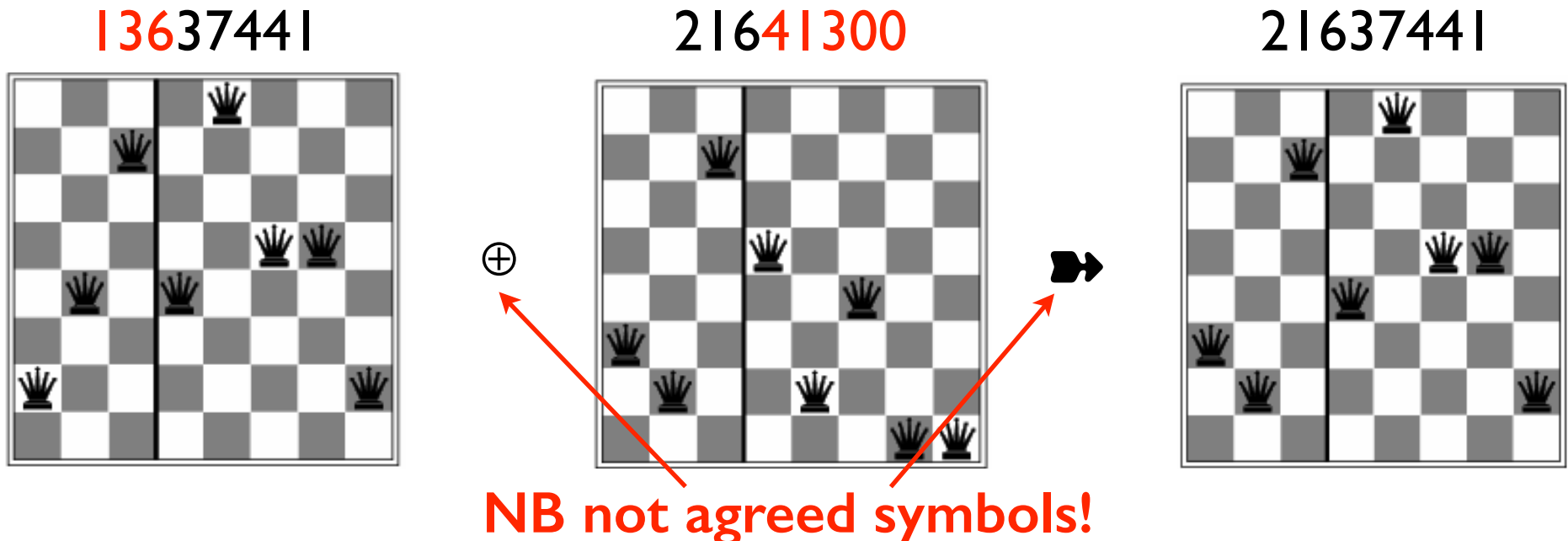
2 21637441

3 21641013

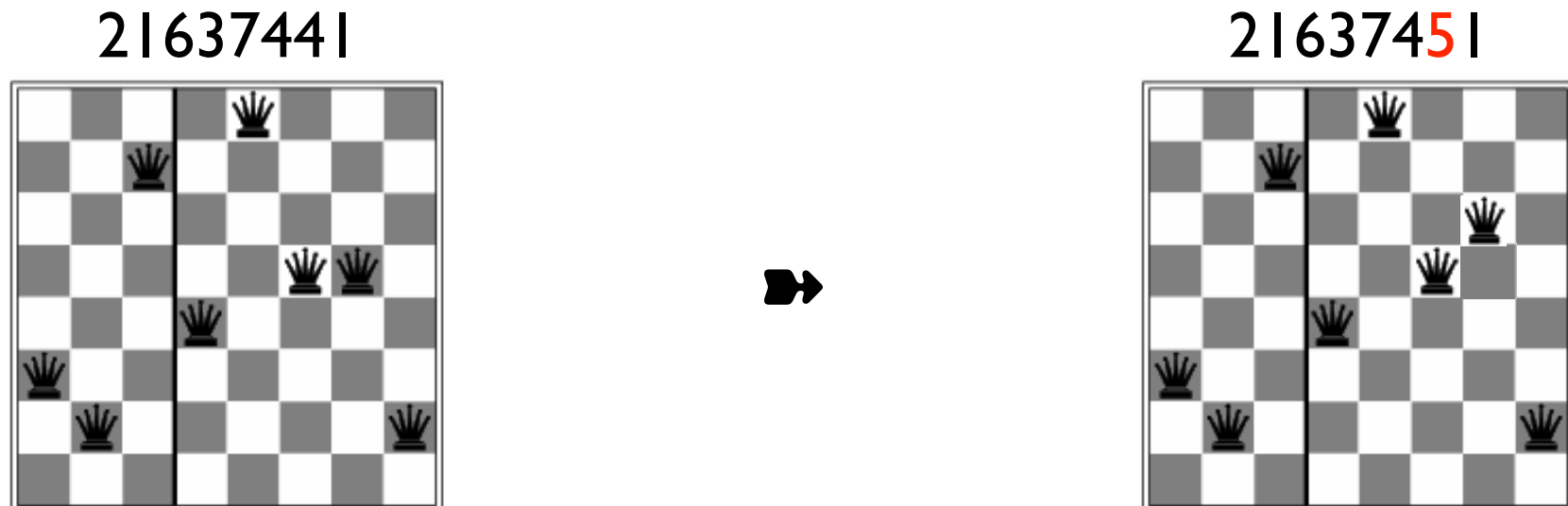
4 13305300

Cross-over leads to new population

- Pairs are selected by one of a range of methods:
 - Roulette-wheel (as in our example)
 - Tournament: rank-influenced selection from a random subset of the population
- Cross-over point for each pair is randomly selected
 - Resulting new chromosomes represent new states



- Cross-over is not enough
 - ▶ if the population does not contain examples that have each bit of the chromosome at both possible values parts of the search space are inaccessible
- So introduce *mutation*
 - ▶ low probability of flipping a random bit at each cross-over step



- Initial Population
 - ▶ Random bit strings – make sure every bit means something
- Selection
 - ▶ Select according to output of fitness function
 - ▶ Various methods (look them up!)
- Cross-over
 - ▶ Select a random point in chromosome
 - ▶ Combine the first part of one example with the second part of another
- Mutation
 - ▶ To introduce possibilities not currently present in the population
- Repeat from “selection” until fitness levels are as desired

Past **Example** Question

- Consider the following 8-bit chromosomes, to be used as the initial state of a very small genetic algorithm.

Bit Number	0	1	2	3	4	5	6	7
Chromosome C0	0	1	1	0	0	0	1	1
Chromosome C1	1	1	0	0	1	0	1	1
Chromosome C2	0	0	0	1	1	0	0	1
Chromosome C3	0	1	1	1	0	0	1	1

- (c) What would be the result of applying crossover to C0 and C2 between bits 4 and 5? [2 marks]
- (d) Given the initial state above, explain why crossover is not enough to explore the entire search space described by this representation. Give a detailed explanation of how this problem is solved in genetic algorithms. [6 marks]

Example solution A

(c) C0: 01100 | 011 C2: 00011 | 001

#1 C0 + #2 C2 = 01100001

#1 C2 + #2 C0 = 00011011

(d) Crossover is not enough to explore the entire search space because in some cases there are positions in the chromosome which, through the whole search space, don't have all possible values. For example, in this search space, the position 7 only appears with value 1, so no matter how many times crossover is applied, position 7 will always be 1. That doesn't help with the variance of the space and the solutions. In genetic algorithms, an operation is applied to try and solve this issue, it is called mutation.

Mutation is a low-probability bit flip at a random position at the chromosome. It happens after the crossover operation, always, if a low-probability threshold is achieved the algorithm changes a random bit in the chromosome, i.e. a bit in a random position

Example Solution B

(c) $C0 = 01100111$

$C2 = 00010001$

The other chromosomes do not change.

(d) Genetic algorithms use mutation to randomly change bits to form compositions not achievable with crossover.