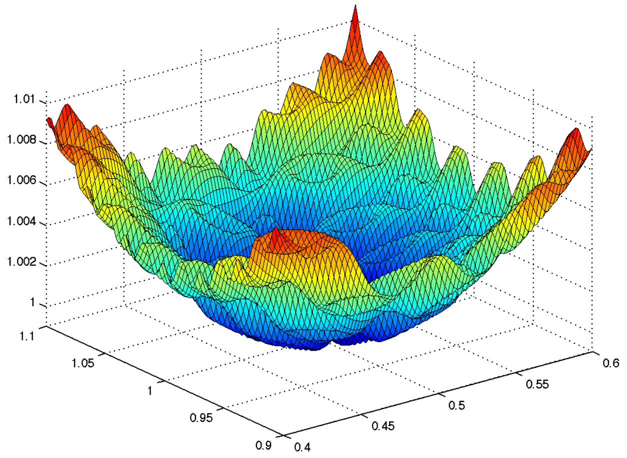
# Local Search Algorithms

Sanja Lazarova-Molnar

# Local search algorithms

- Some types of search problems can be formulated in terms of **optimization**
  - We don’t have a start state, don’t care about the path to a solution
  - We have an **objective function** that tells us about the quality of a possible solution, and we want to find a good solution by minimizing or maximizing the value of this function

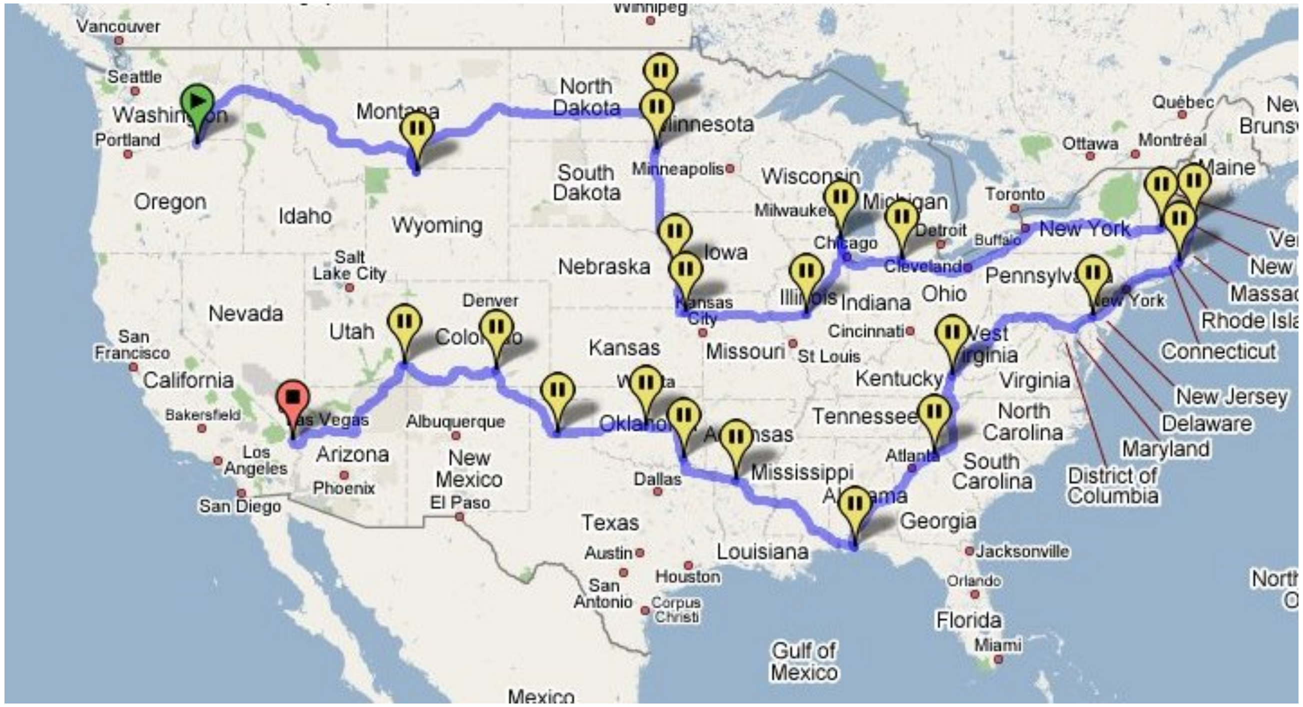


## Approaches

- Hill Climbing
- Simulated Annealing
- Genetic Algorithms

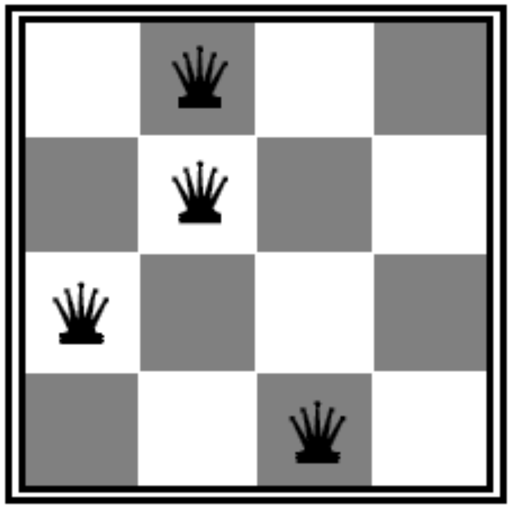
# Example: Traveling salesman problem

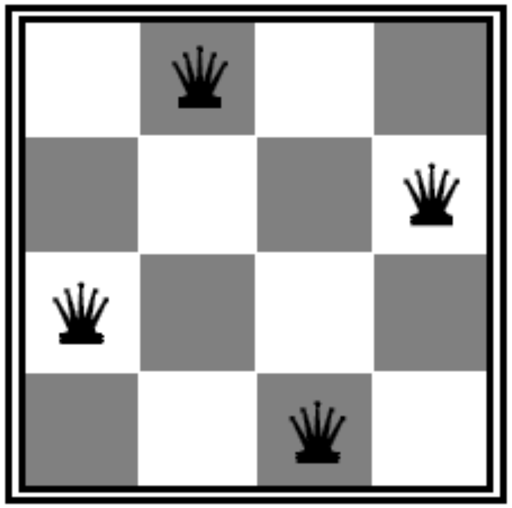
- Find the shortest tour connecting a given set of cities
- **State space:** all possible tours
- **Objective function:** length of tour



# Example: *n*-queens problem

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- **State space:** all possible *n*-queen configurations
- What's the **objective function**?
  - Number of pairwise conflicts



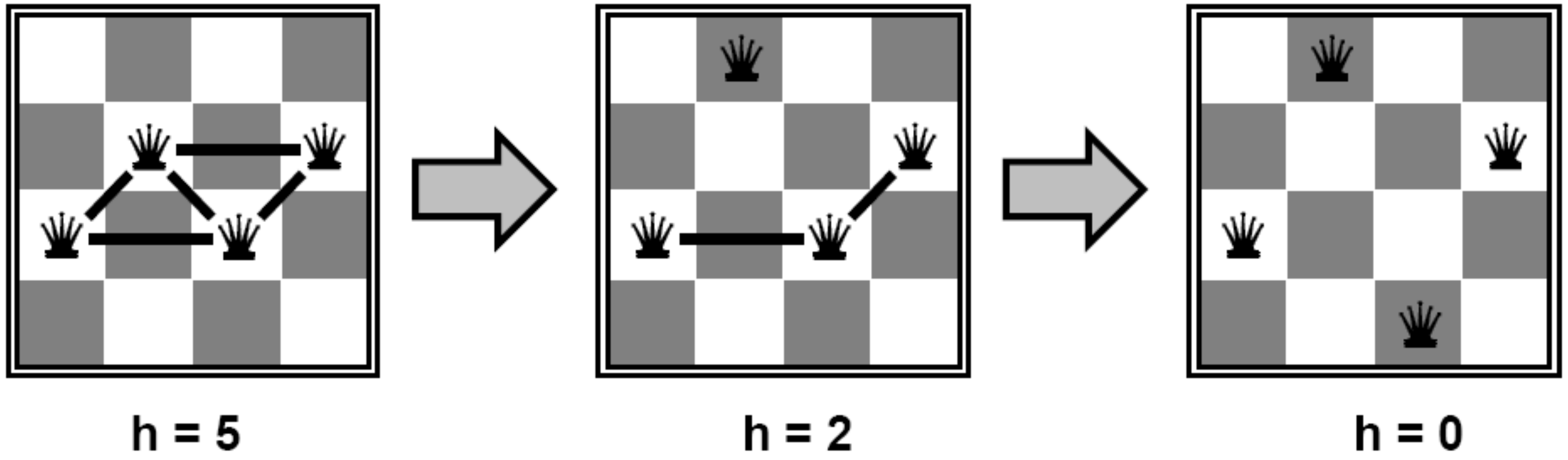


## Hill-climbing (greedy) search

- Idea: keep a single “current” state and try to locally improve it
- “Like climbing mount Everest in thick fog with amnesia”

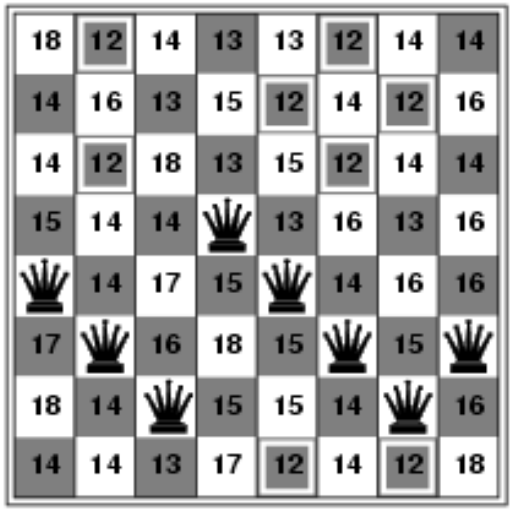
# Example: *n*-queens problem

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- **State space:** all possible *n*-queen configurations
- **Objective function:** number of pairwise conflicts
- What's a possible local improvement strategy?
  - Move one queen within its column to reduce conflicts



# Example: *n*-queens problem

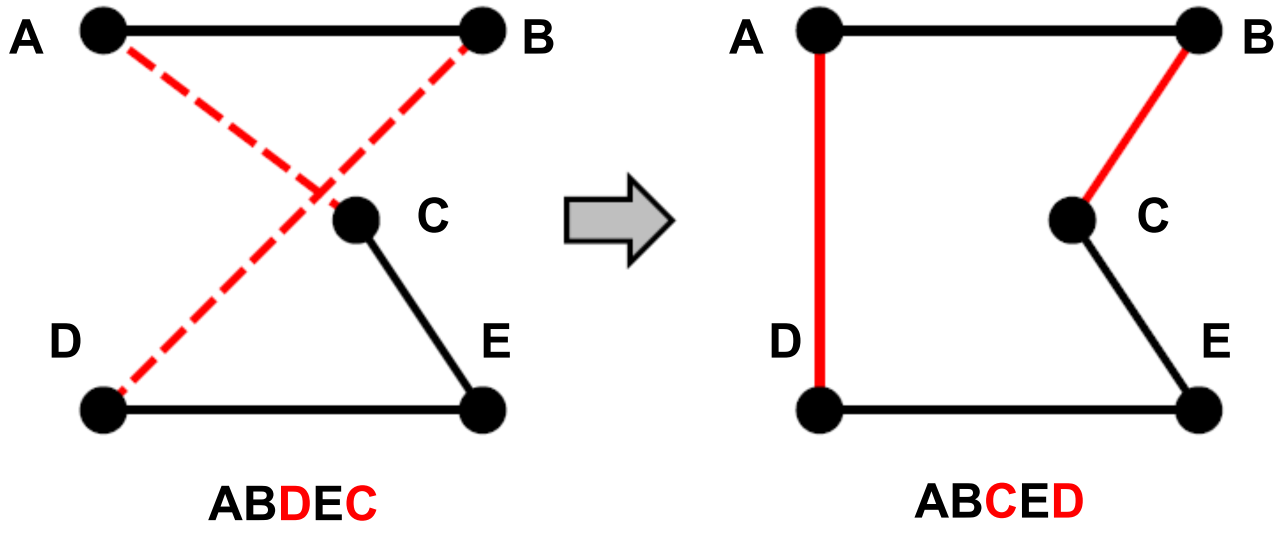
- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- **State space:** all possible *n*-queen configurations
- **Objective function:** number of pairwise conflicts
- What's a possible local improvement strategy?
  - Move one queen within its column to reduce conflicts



*h* = 17

# Example: Traveling Salesman Problem

- Find the shortest tour connecting n cities
- **State space:** all possible tours
- **Objective function:** length of tour
- What's a possible local improvement strategy?
  - Start with any complete tour, perform pairwise exchanges



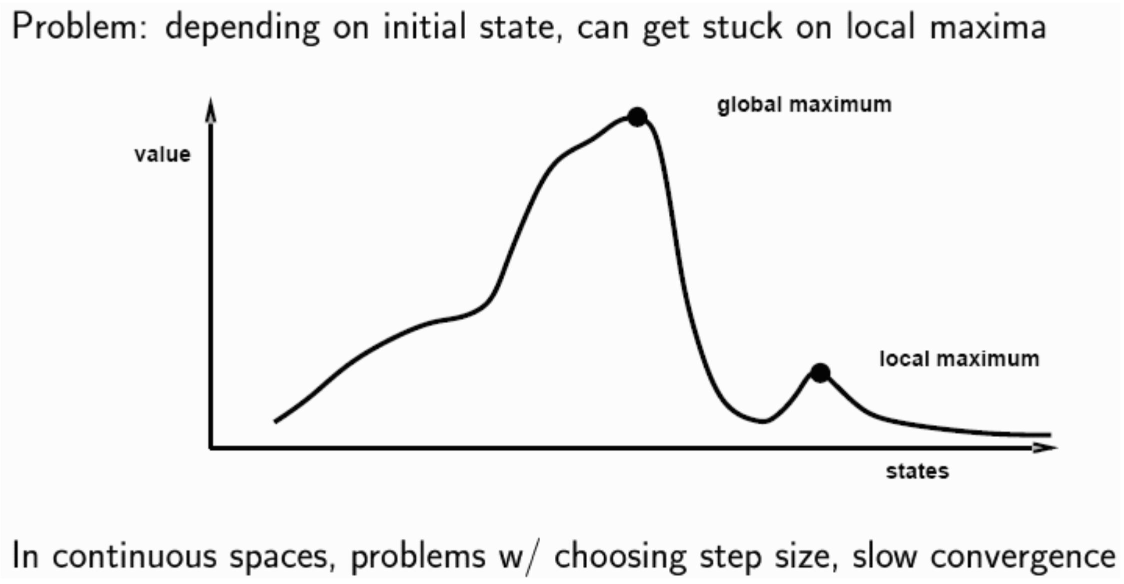
# Hill-climbing (greedy) search

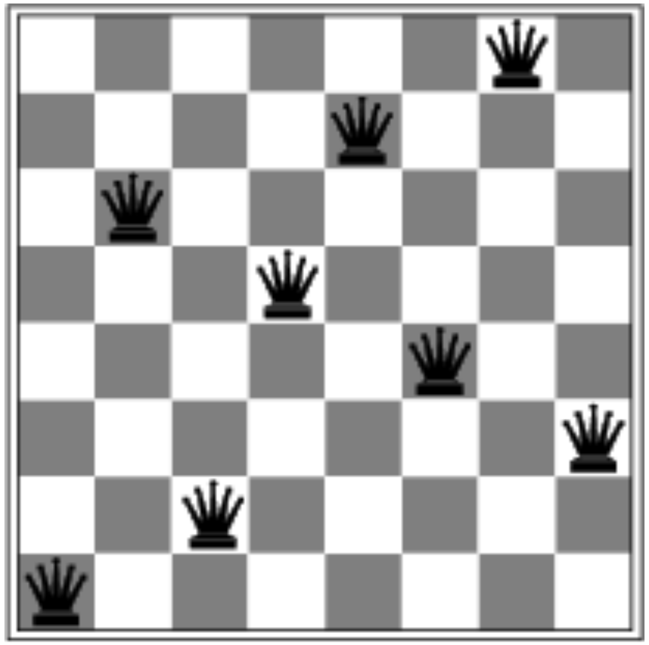
- Initialize *current* to starting state
- Loop:
  - Let *next* = highest-valued successor of *current*
  - If value(*next*) < value(*current*) return *current*
  - Else let *current* = *next*

- Variants: choose first better successor, randomly choose among better successors

# Hill-climbing search

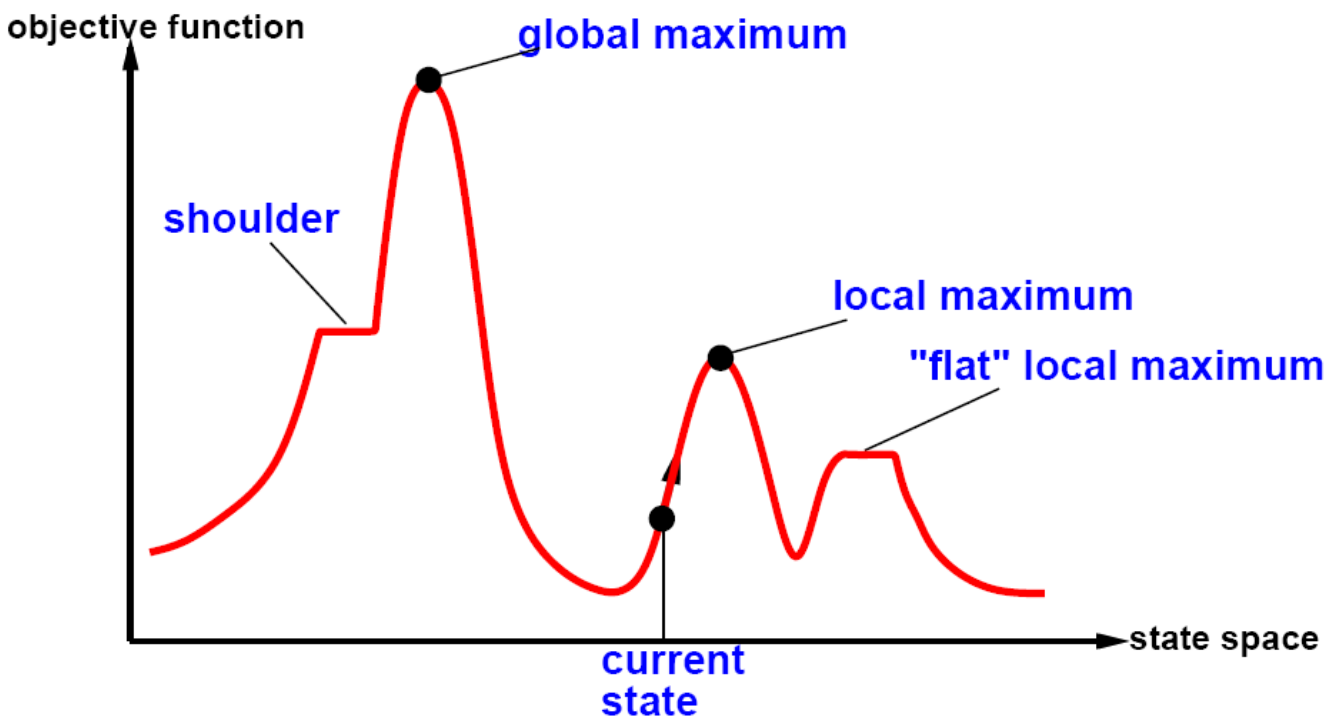
- Is it complete/optimal?
  - No – can get stuck in local optima (peak higher than neighbors but lower than global maximum)
  - Example: local optimum for the 8-queens problem





*h* = 1

# The state space “landscape”

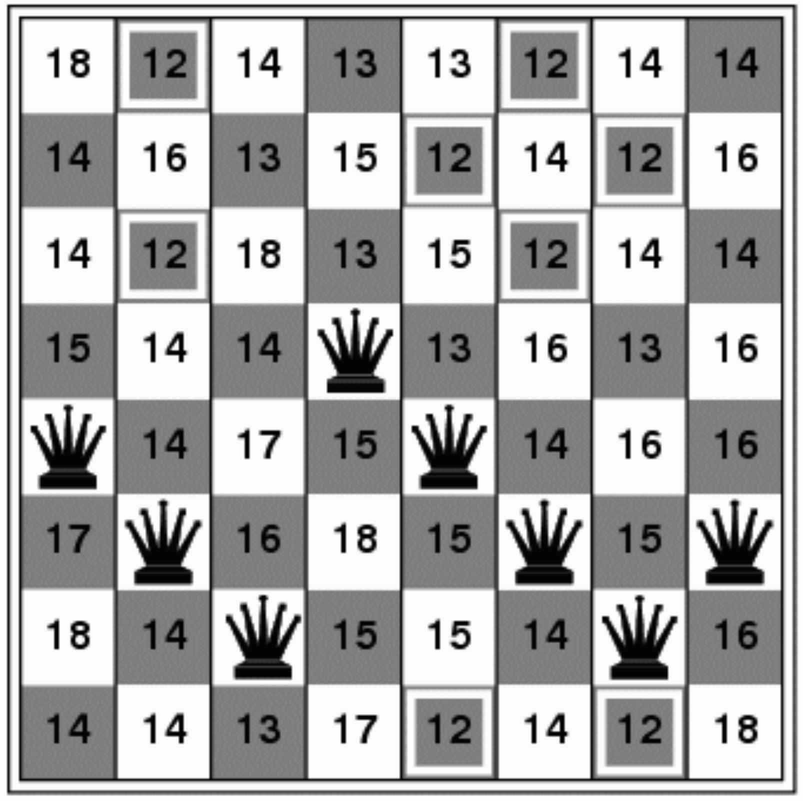


- How to escape local maxima?
  - Random restart hill-climbing

    iteratively does hill-climbing, each time with a random initial condition . The best is kept: if a new run of hill climbing produces a better than the stored state, it replaces the stored state.

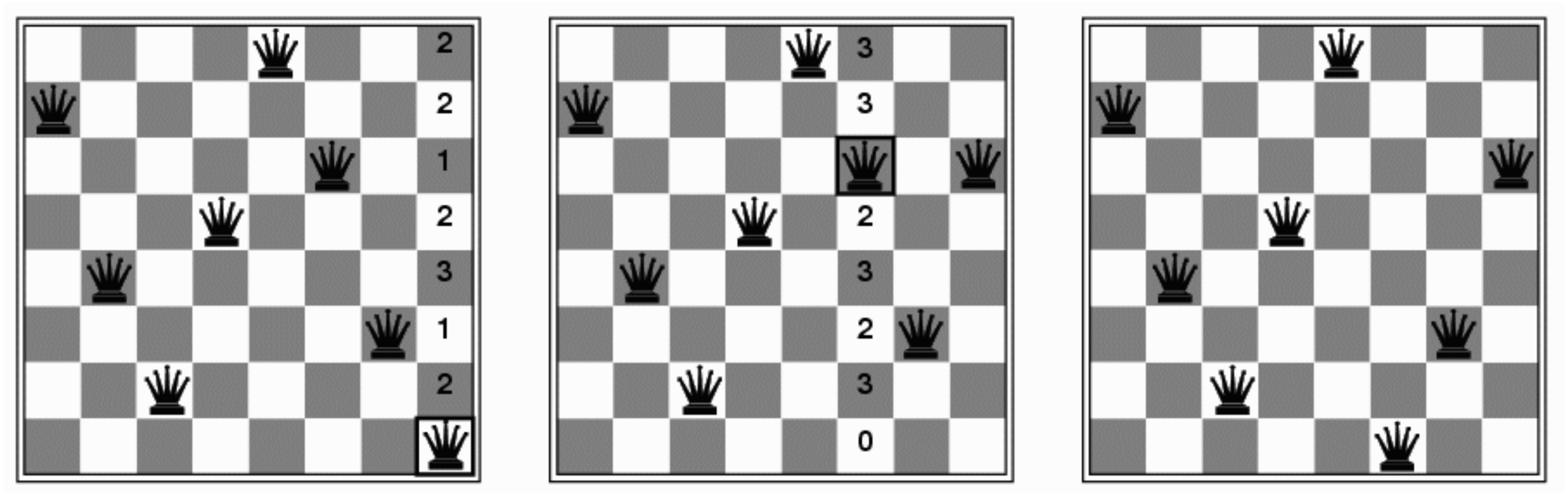
# Example (hill-climbing)

- The figure at right has a heuristic cost estimate value of h=17, the number of conflicting pairs of queens.
- The figure gives the estimate of each successor state.
- The successor function returns all possible 8*7=56 states reachable after a given move in a column.



# Following figures

- Queen in lower right of first figure in conflict with 2 others by moving up one row. Moving to a row with 1 conflict would be a local minima.
- In second figure, queen is moved to local minima 0 which turns out to be a global minimum.
- Last figure has total number of conflicts h=0, a global minimum.



## Simulated annealing search

- Hill-climbing algorithm never makes downhill move - always incomplete
- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency
  - Probability of taking downhill move decreases with number of iterations, steepness of downhill move
  - Controlled by *annealing schedule*
- Inspired by annealing process, as part of the tempering of glass, metal

# Simulated annealing search

```
• Initialize current to starting state
• for i = 1 to ∞
      let next = random successor of current
      let Δ = value(next) – value(current)
      if Δ > 0 then let current = next
      else let current = next with probability exp(Δ/T(i))
```

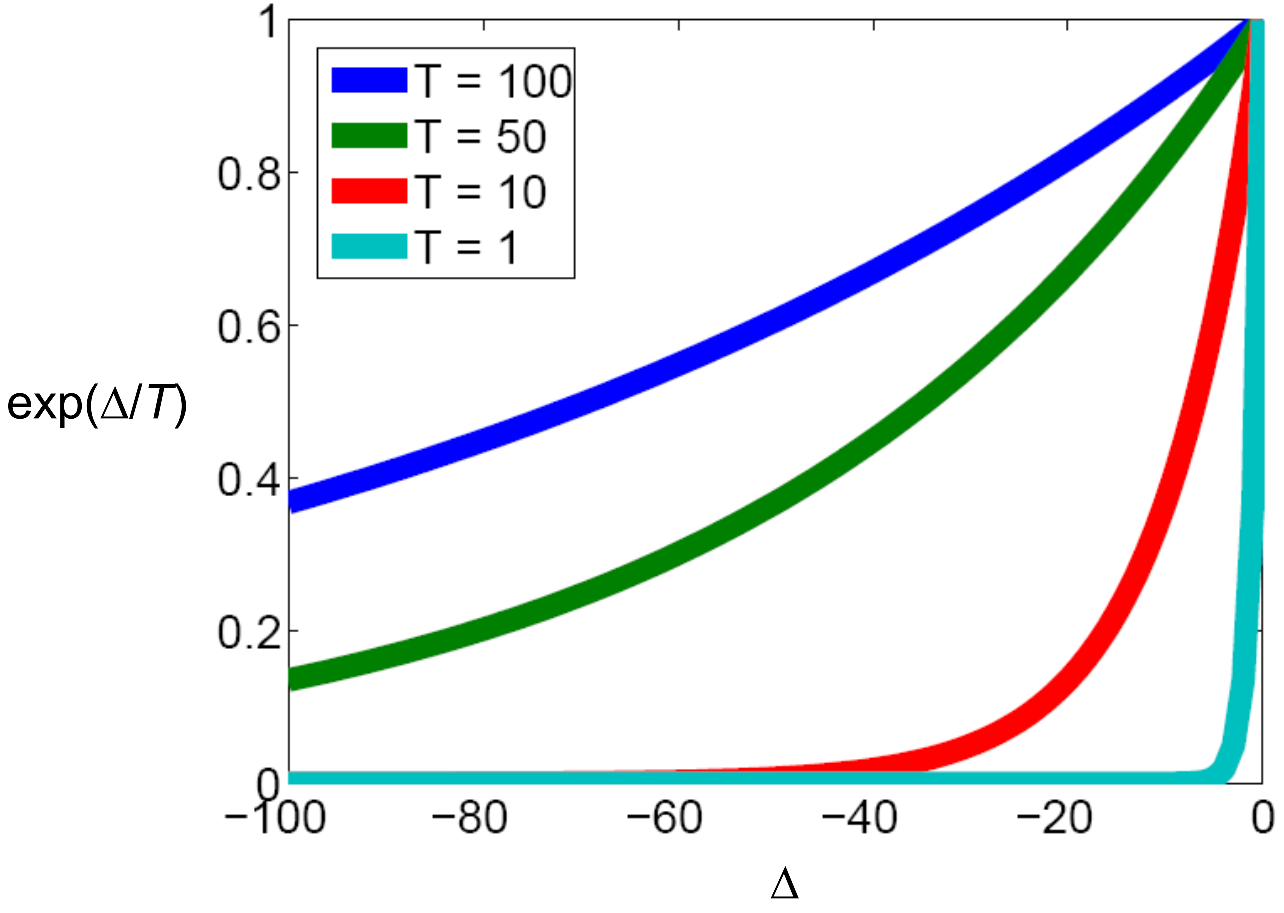
- *T* gradually decreased to 0 over time *t.*

  Picks random rather than best state move as in hill-climbing.

  If move is improvement over current state, always accepted, otherwise accepted with probability < 1.

  Probability decreases exponentially when move is worse than current and as *T* decreases over time; bad moves more likely to be accepted early.

# Effect of temperature



# Simulated annealing search

- One can prove: If temperature decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching one
- However:
  - This usually takes impractically long
  - The more downhill steps you need to escape a local optimum, the less likely you are to make all of them in a row
- More modern techniques: general family of *Markov Chain Monte Carlo* (MCMC) algorithms for exploring complicated state spaces

# Genetic Algorithms - History

- Pioneered by John Holland in the 1970’s
- Got popular in the late 1980’s
- Based on ideas from Darwinian Evolution
- Can be used to solve a variety of problems that are not easy to solve using other techniques

# Evolution in the real world

- Each cell of a living thing contains ***chromosomes*** - strings of *DNA*
- Each chromosome contains a set of ***genes*** - blocks of DNA
- Each gene determines some aspect of the organism (like eye colour)
- A collection of genes is sometimes called a ***genotype***
- A collection of aspects (like eye colour) is sometimes called a ***phenotype***
- Reproduction involves recombination of genes from parents and then small amounts of ***mutation*** (errors) in copying
- The ***fitness*** of an organism is how much it can reproduce before it dies
- Evolution based on “survival of the fittest”

## To start with …

- Suppose you have a problem
- You don’t know how to solve it
- What can you do?

## A dumb solution

A “blind generate and test” algorithm:

```
Repeat
    Generate a random possible solution
    Test the solution and see how good it is
Until solution is good enough
```

# Can we use this dumb idea?

- Sometimes - yes:
  - if there are only a few possible solutions
  - and you have enough time
  - then such a method *could* be used
- For most problems - no:
  - many possible solutions
  - with no time to try them all
  - so this method *can not* be used

## A “less-dumb” idea (GA)

```
Generate a set of random solutions
Repeat
    Test each solution in the set (rank them)
    Remove some bad solutions from set
    Duplicate some good solutions
        make small changes to some of them
Until best solution is good enough
```

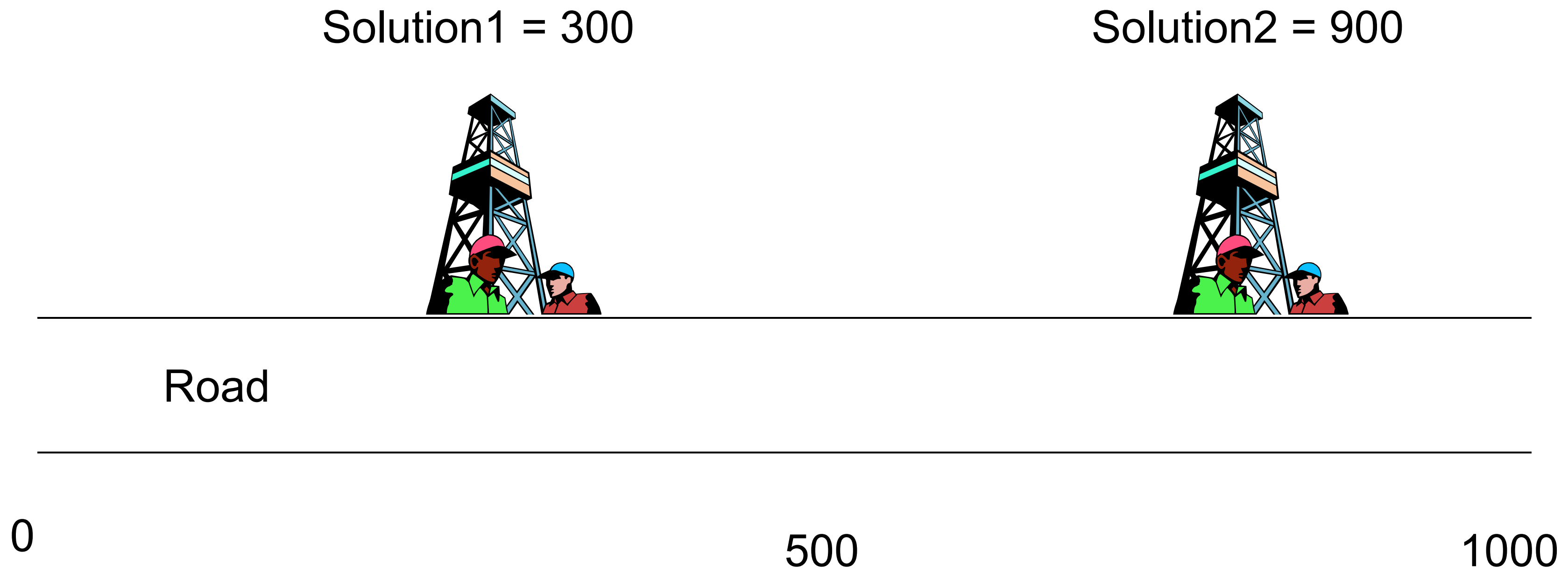
# How do you encode a solution?

- Obviously this depends on the problem!
- GA’s *often* encode solutions as fixed length “bitstrings” (e.g. 101110, 111111, 000101)
- Each bit represents some aspect of the proposed solution to the problem
- For GA’s to work, we need to be able to “test” any string and get a “score” indicating how “good” that solution is

## Silly Example - Drilling for Oil

- Imagine you had to drill for oil somewhere along a single 1km desert road
- Problem: choose the best place on the road that produces the most oil per day
- We could represent each solution as a position on the road
- Say, a whole number between [0..1000]

# Where to drill for oil?



# Digging for Oil

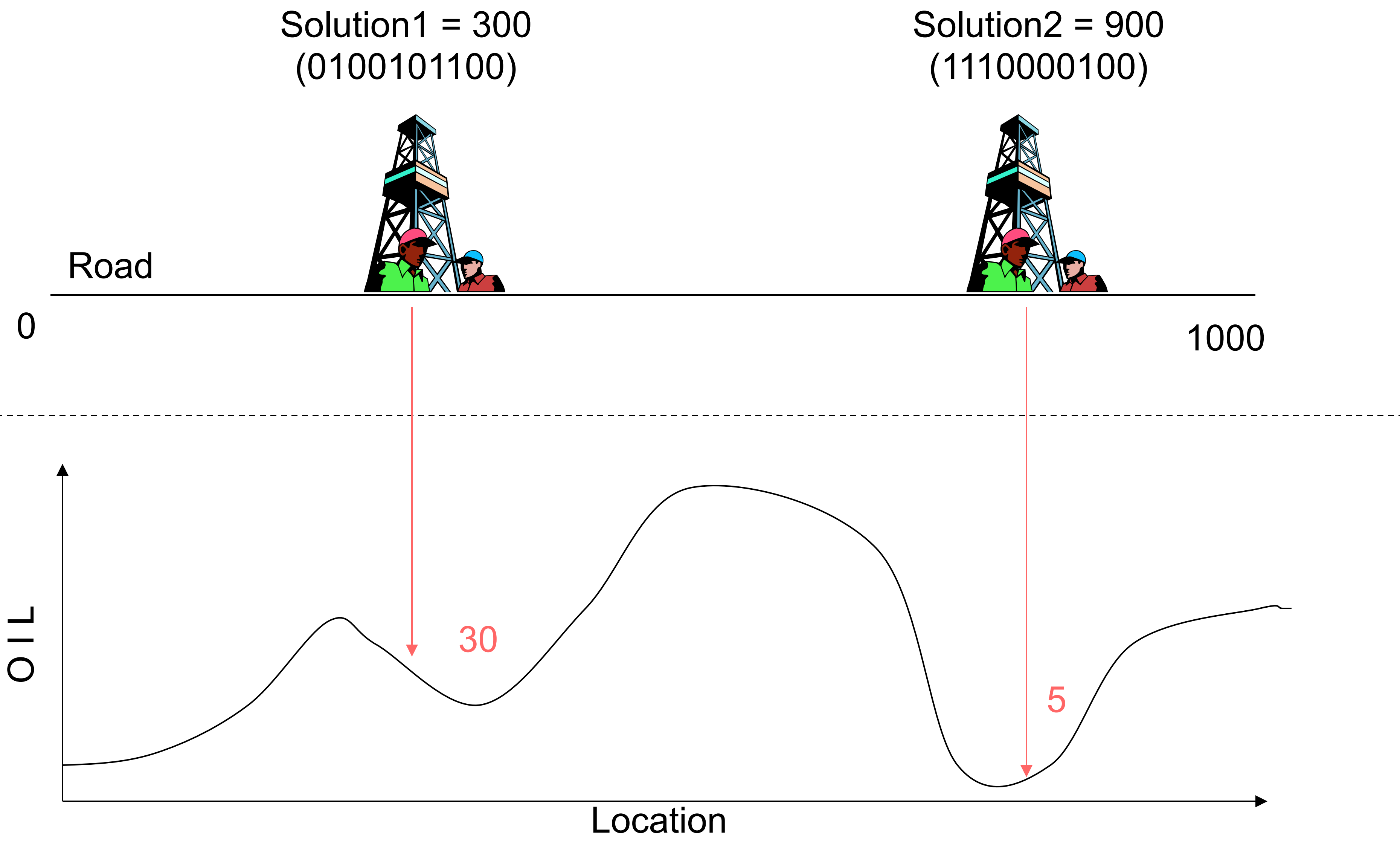
- The set of all possible solutions [0..1000] is called the *search space* or *state space*
- In this case it’s just one number but it could be many numbers or symbols
- Often GA’s code numbers in binary producing a bitstring representing a solution
- In our example we choose 10 bits which is enough to represent 0..1000

# Convert to binary string

| | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|
| 900 | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** |
| 300 | **0** | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **0** | **0** |
| 1023 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

In GA's these encoded strings are sometimes called "*genotypes*" or "*chromosomes*" *and the individual bits are sometimes called "genes"*

# Drilling for Oil



## Summary

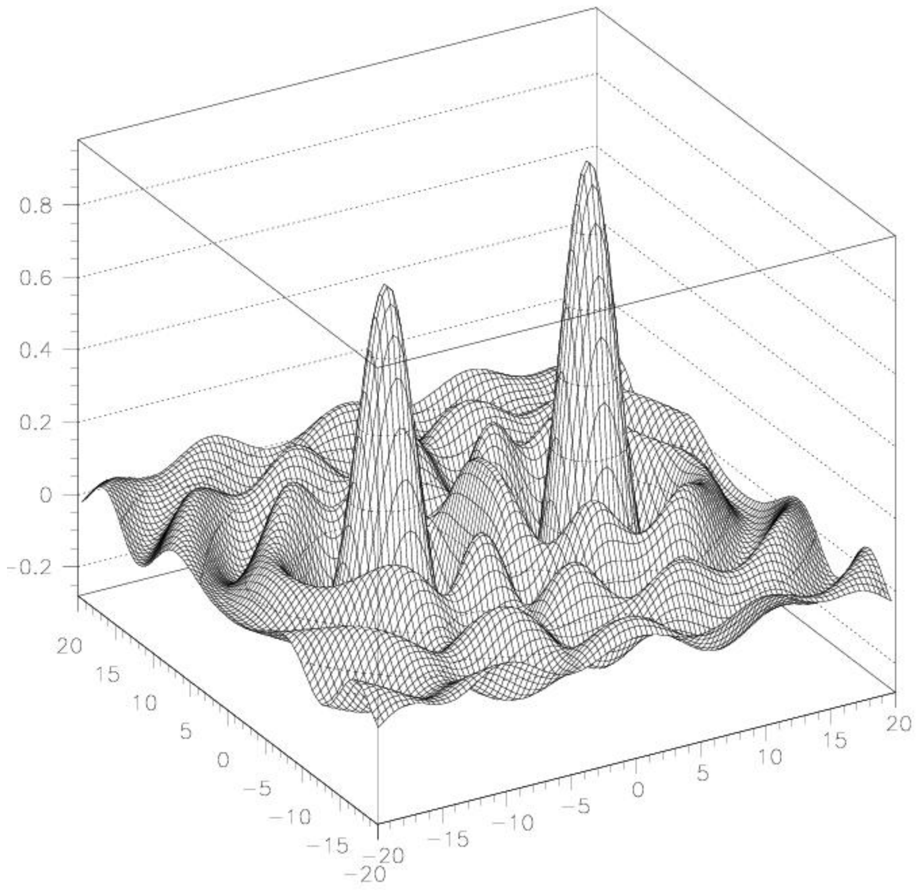
We have seen how to:

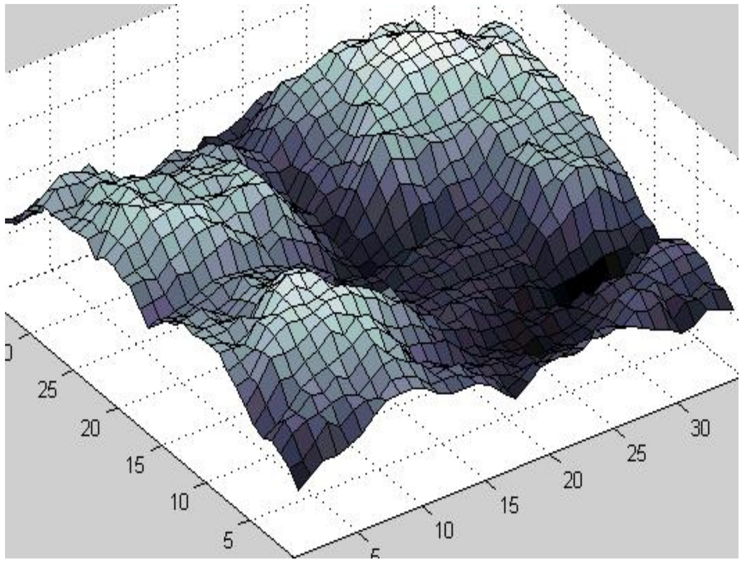
- represent possible solutions as a number
- encoded a number into a binary string
- generate a score for each number given a *function* of “how good” each solution is - this is often called a *fitness function*
- Our silly oil example is really optimisation over a function f(x) where we adapt the parameter x

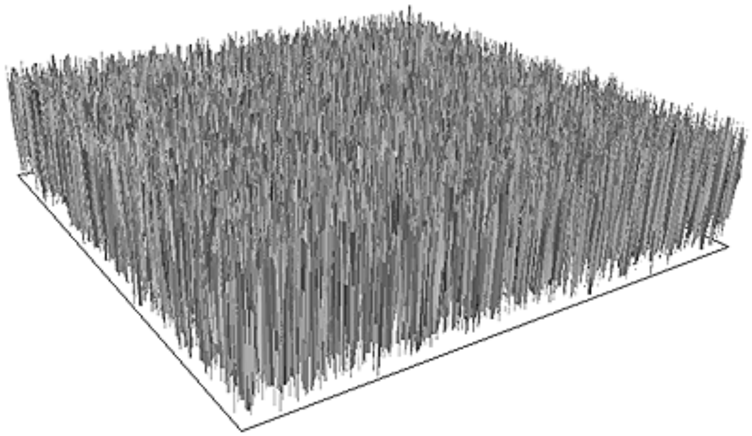
## Search Space

- For a simple function f(x) the search space is one dimensional.
- By encoding several values into the chromosome many dimensions can be searched e.g. two dimensions f(x,y)
- Search space an be visualised as a surface or *fitness landscape* in which fitness is height
- A GA tries to move the points to better places (higher fitness) in the space

# Fitness landscapes







## Search Space

- Obviously, the nature of the search space dictates how a GA will perform
- A completely random space would be bad for a GA
- Also GA’s can get stuck in local maxima if search spaces contain lots of these
- Generally, spaces in which small improvements get closer to the global optimum are good

# Back to the (GA) Algorithm

```
Generate a set of random solutions
Repeat
    Test each solution in the set (rank them)
    Remove some bad solutions from set
    Duplicate some good solutions
        make small changes to some of them
Until best solution is good enough
```

## Adding Reproduction - Crossover

- Although it may work for simple search spaces - our algorithm is still very simple
- Relies on random mutation to find a good solution
- Introducing reproduction - better results

## Adding Reproduction - Crossover

- Two high scoring “parent” bit strings (*chromosomes)* are selected and with some probability (crossover rate) combined
- Producing two new *offspring* (bit strings)
- Each offspring may then be changed randomly (*mutation*)

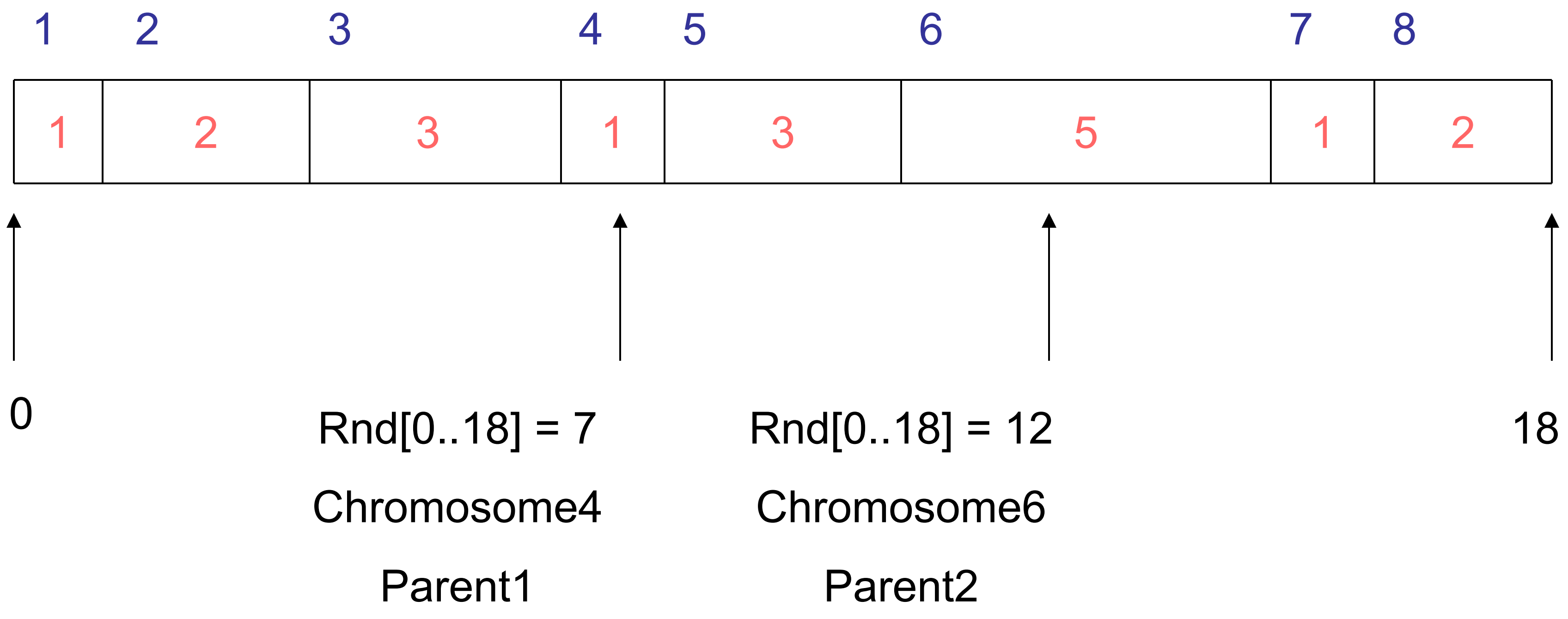
## Selecting Parents

- Many schemes are possible, goal: better scoring chromosomes more likely to be selected
- Score is often termed the *fitness*
- “Roulette Wheel” selection can be used:
  - Add up the fitness's of all chromosomes
  - Generate a random number R in that range
  - Select the first chromosome in the population that - when all previous fitness’s are added - gives you at least the value R

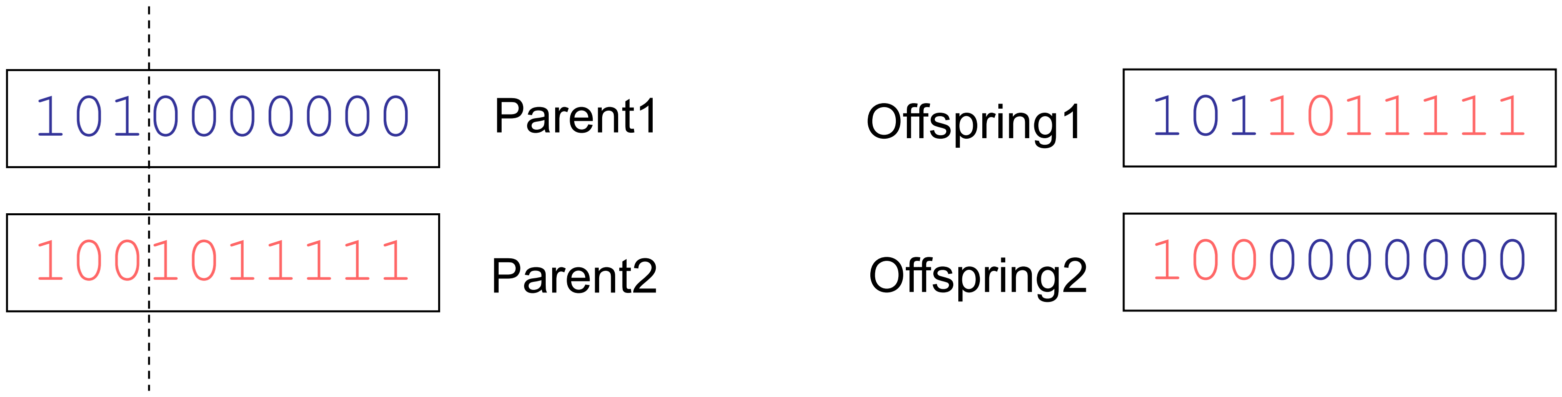
# Example population

| No. | Chromosome | Fitness |
|---|---|---|
| 1 | 1010011010 | 1 |
| 2 | 1111100001 | 2 |
| 3 | 1011001100 | 3 |
| 4 | 1010000000 | 1 |
| 5 | 0000010000 | 3 |
| 6 | 1001011111 | 5 |
| 7 | 0101010101 | 1 |
| 8 | 1011100111 | 2 |

# Roulette Wheel Selection



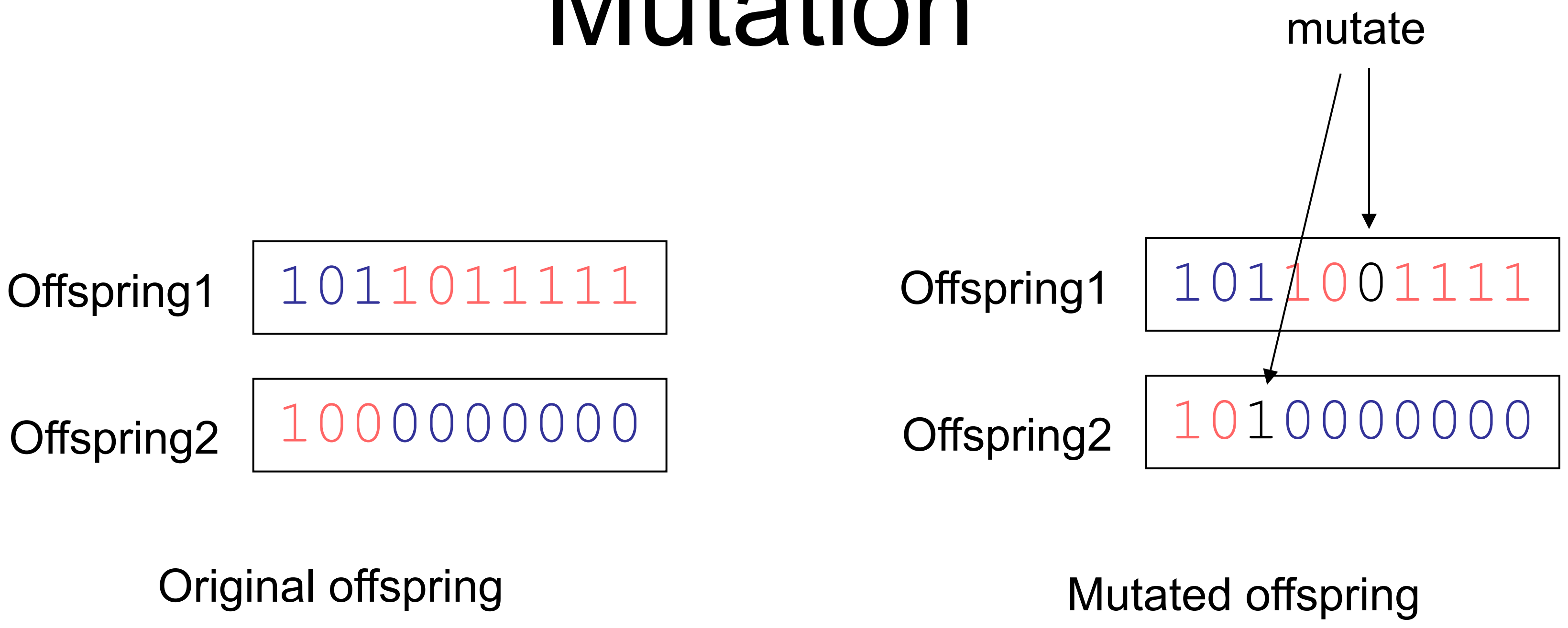
# Crossover - Recombination



Crossover single point - random

With some high probability (*crossover rate*) apply crossover to the parents. (*typical values are 0.8 to 0.95*)

# Mutation



With some small probability (the *mutation rate*) flip each bit in the offspring (*typical values between 0.1 and 0.001*)

## Back to the (GA) Algorithm

```
Generate a population of random chromosomes
Repeat (each generation)
    Calculate fitness of each chromosome
    Repeat
        Use roulette selection to select pairs of parents
        Generate offspring with crossover and mutation
    Until a new population has been produced
Until best solution is good enough
```

# Many Variants of GA

- Different kinds of selection (not roulette)
  - Tournament
  - Elitism, etc.
- Different recombination
  - Multi-point crossover
  - 3 way crossover etc.
- Different kinds of encoding other than bitstring
  - Integer values
  - Ordered set of symbols
- Different kinds of mutation

## Many parameters to set

- Any GA implementation needs to decide on a number of parameters: Population size (N), mutation rate (m), crossover rate (c)
- Often these have to be “tuned”, based on results obtained - no general theory to deduce good values
- Typical values might be: N = 50, m = 0.05, c = 0.9

# Genetic algorithms-summarized

- Start with randomly generated population of valid candidate states
- For generation in (0..n)
  - Select 2 states (parents) randomly from population for reproduction based on *fitness* of each state. The more fit a state, the more likely selected to reproduce.
  - Reproduce a new state (child) by combining random parts of the 2 selected states (crossover)
  - Mutate a random number of new states to allow exploration other possible states
  - Population = new states
- Best solution is fittest of population states

# Example

- A trivial problem is to determine the greatest 3-bit binary number. The diagram at right illustrates the state space.
- **Representation:** a state is represented by a string. The 3 bits could be represented by strings such as: (0,0,1) etc.
- **Initial population:** Some random set of states, for example: [(1,0,0) (0,1,0) (0,1,0) (0,0,0)]
- **Fitness function:** determines fitness of each state; returns the integer value of the 3-bit number, 0 to 7.
- **Selection:** individuals of each generation are randomly selected using a fitness ratio, their percentage contribution to the total fitness of the population.

# Example ctd.

For the first generation:

| Genes | Fitness | Fitness ratio |
|---|---|---|
| (1,0,0) | 4 | 50% |
| (0,1,0) | 2 | 25% |
| (0,1,0) | 2 | 25% |
| (0,0,0) | 0 | 0% |

a total fitness 4+2+2+0=8. (1,0,0) selected with 0.5, (0,1,0) with 0.25, (0,1,0) with 0.25 and (0,0,0) with 0.0 probability.

Randomly selected pairs using fitness ratio: [(1,0,0) (0,1,0)], [(1,0,0) (1,0,0)], [(0,1,0) (0,1,0)], [(0,1,0) (1,0,0)]

# GA ctd.

**Reproduce:** combine selected state pairs at some random point using crossover. The first state is copied up to the crossover point, then the second state is copied from there on.

[(1,0,0) (0,1,0)] combined at bit 1 producing (1,1,0)

[(1,0,0) (1,0,0)] combined at bit 0 producing (1,0,0)

[(0,1,0) (0,1,0)] combined at bit 2 producing (0,1,0)

[(0,1,0) (1,0,0)] combined at bit 1 producing (0,0,0)

## GA ctd.

Note that the higher order bits contribute more to our definition of fitness; combining a high and low fitness state could produce a lower fitness result.

Stagnation: it is important that less fit individuals occasionally are selected, though all individuals are selected at rate proportional to their fitness. This helps ensure populations do not stagnate by constantly selecting from the same parent states. For example, if only the most fit state, (1, 0, 0), were selected the optimal state of (1, 1, 1) would never be found.

**Mutation:** Because bit 0 is 0 throughout the population, no state with bit 0 a 1 can ever be explored using crossover alone. Change a random element in a state representation at some specified probability. For example:

(0,1,0) mutated to (0,1,1)

# GA ctd.

**New Population:** [(1,1,0), (1,0,0), (0,1,1), (0,0,0)] has total fitness of 6+4+3+0=13

| Genes | Fitness | Fitness ratio |
|---|---|---|
| (1,1,0) | 6 | 46% |
| (1,0,0) | 4 | 31% |
| (0,1,1) | 3 | 23% |
| (0,0,0) | 0 | 0% |

**Terminate generation or fit enough:** Terminate if individual fit enough or number of generations reached.