Local search
(CK0031/CK0248)

Francesco Corona

Department of Computer Science
Federal University of Ceará, Fortaleza

Beyond search

So far, a single category of problems

- Observable, deterministic and known environments

The solution is a sequence of actions

What if some of these assumptions are relaxed?

- ⇝ Algorithms that perform pure **local search** in a state space
- ⇝ Evaluation and modification of one or more current states

Alternative to systematic path exploration from a start state

Beyond search (cont.)

Algorithms suitable for problems in which all that matters is solution state

- The path cost to reach it is not the priority

Algorithms for continuous state and action spaces

- **Local search in continuous space**

⇝ **Numerical optimisation**

Beyond search (cont.)

We relax the assumptions of determinism and observability

- The agent cannot predict exactly what percept it will receive
- The agent will also need to keep track of the states it might be in
- (because of partial observability)

Beryond search (cont.)

We investigate **online search**

- The agent is faced with a state space that is initially unknown

⇝ The state space must be explored

Local search and optimisation

Local search and optimisation

Search algorithms so far are designed to explore spaces systematically

- Systematicity

Keep paths in memory and record which alternatives were explored

- At each point along path

A path to the goal also constitutes a solution to the problem

- In many problems, path to goal is irrelevant

Local search and optimisation

Example

In the 8-queens problem what matters is the final configuration

- The order in which the queens are added is irrilevant

The same general property holds for many important applications

- Integrated-circuit design
- Factory-floor layout
- Job-shop scheduling
- Automatic programming
- Telecom network optimisation
- Vehicle routing
- Portfolio management

Local search and optimisation (cont.)

Suppose that the path to goal does not matter

- We consider a different class of algorithms
- Algorithms that do not worry about paths at all

Definition

Local search algorithms operate using a single current node

- *(rather than multiple paths)*

Generally, these algorithm move only to neighbours of that node

Typically, the paths followed by the search are not retained

Local search and optimisation (cont.)

Local search algorithms are not systematic

They have two key advantages

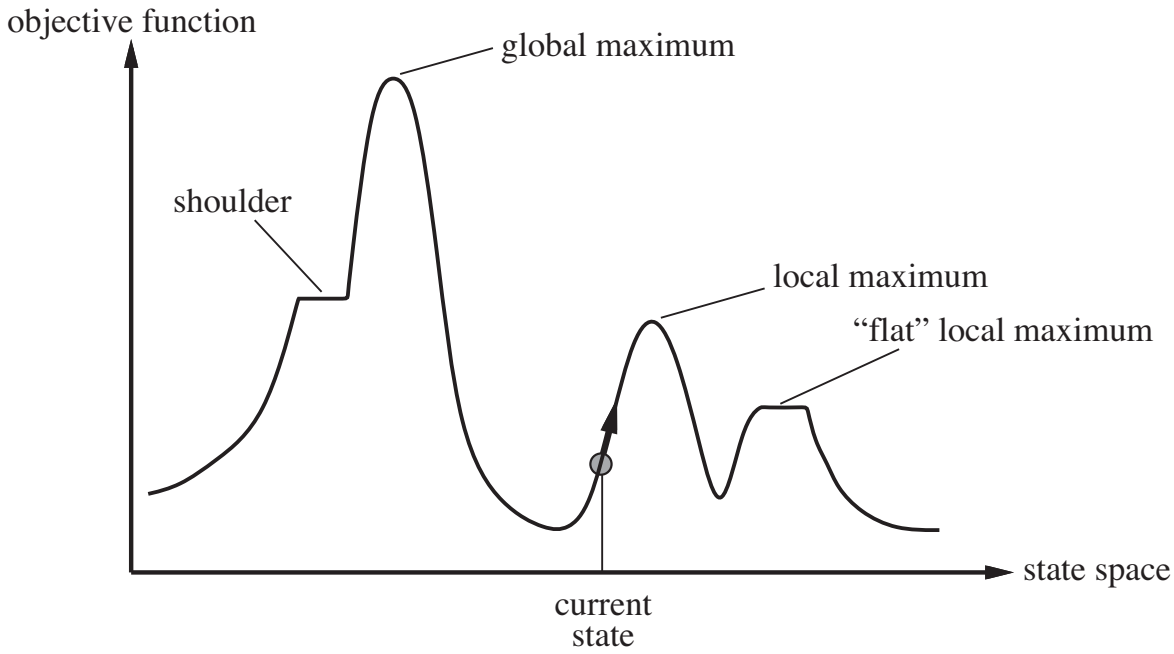
- They use very little memory (usually a constant amount)
- They can find okay solutions in continuous state spaces
- (for which systematic algorithms are unsuitable)

Local search is valid for solving pure **optimisation problems**

- Find the best state, according to an **objective function**

Local search and optimisation (cont.)

To understand local search, consider the state-space landscape



The landscape

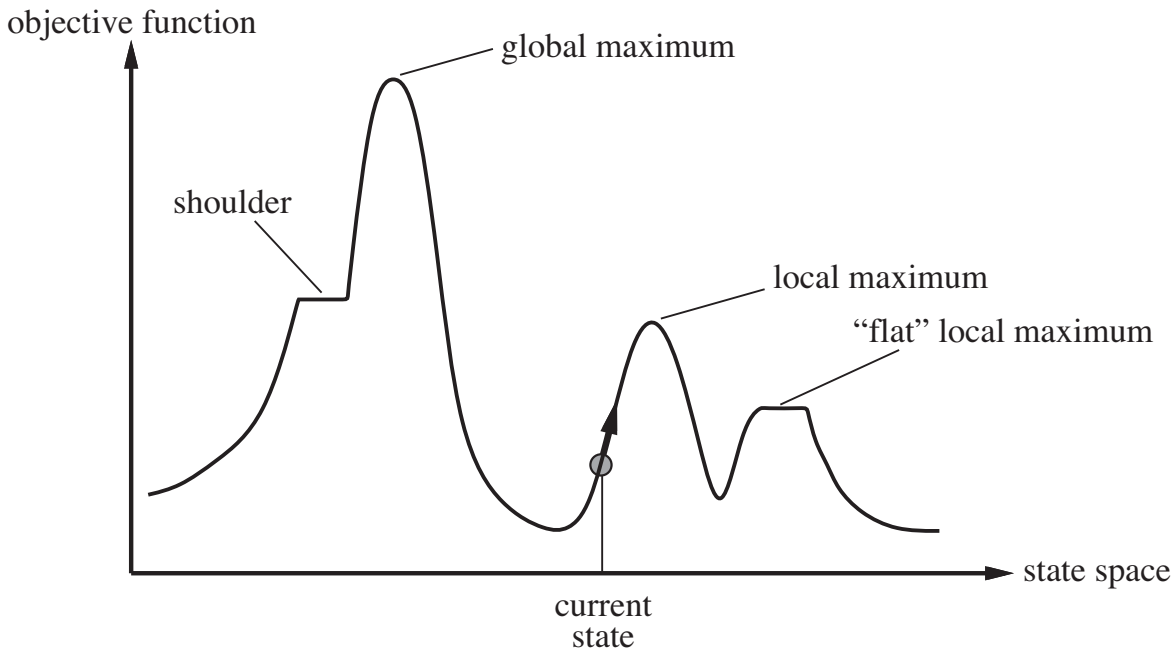
- A 'location' (the state)
- An 'elevation' (cost/objective function)

Elevation is the value of the heuristic cost function or objective function

- Various states are characterised by different frunction values

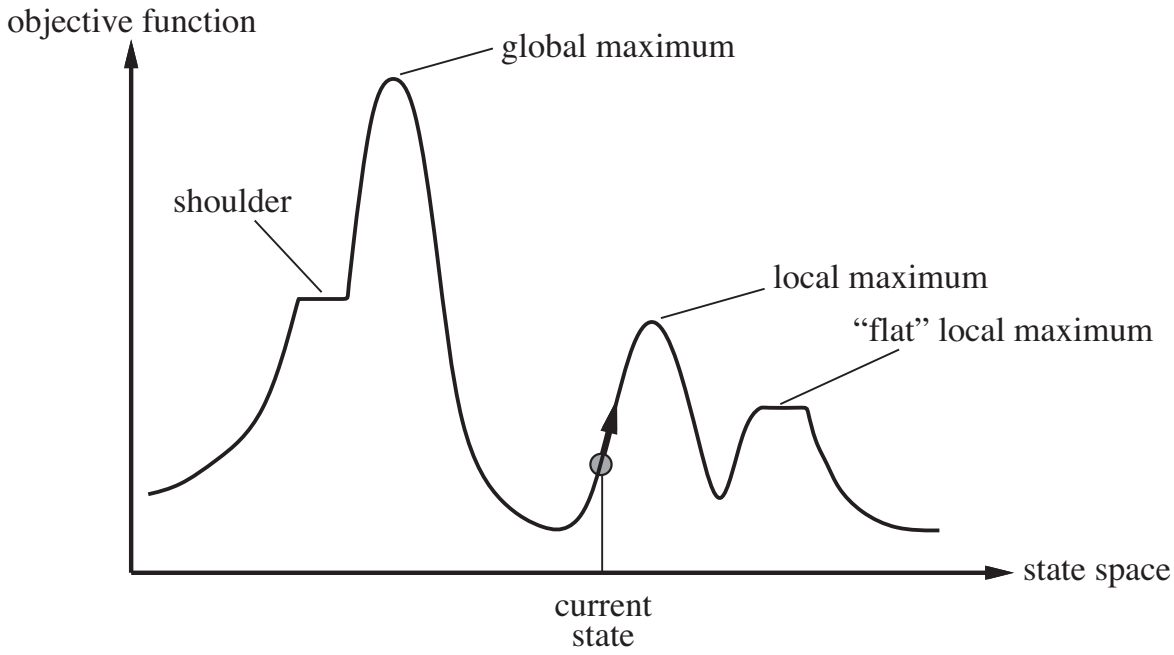
Local search and optimisation (cont.)

Local search algorithms explore this landscape



⇝ Current state is modified to improve the objective

Local search and optimisation (cont.)



Definition

The target/goal

If elevation corresponds to an objective function

⇝ *Them, the highest peak is the* ***global maximum***

If elevation corresponds to a cost function

⇝ *Then, , the lowest valley is the* ***global minimum***

Conversion from one to the other just by inserting a minus sign

Local search and optimisation (cont.)

Definition

*A **complete** local search algorithm always finds a goal, if one exists*

*An **optimal** algorithm always finds a global minimum/maximum*

Hill-climbing search

Local search and optimisation

Hill-climbing search

The **hill-climbing search** algorithm (**steepest-ascent version**)

- It is a loop that continually moves up-hill
- Always in the direction of increasing value

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*

At each step, current node is replaced by the best of its neighbours

- It ends at a 'peak' where no neighbour has a higher value

Hill-climbing search (cont.)

The algorithm does not keep a search tree

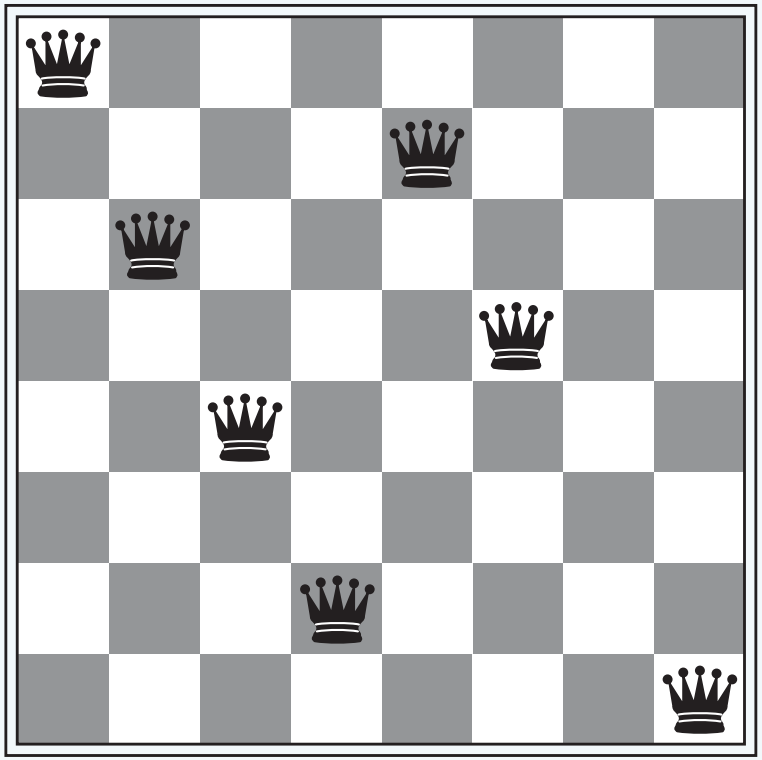
- Data structure for current node only records state and objective's value

There is no ahead looking beyond the immediate neighbours of a state

Hill-climbing search (cont.)

Example

Local search algorithms often use a **complete-state formulation**



- Each state has 8 queens on the board, one per column

Hill-climbing search (cont.)

The successors of a state

- ⇝ All possible states generated by moving a single queen
- ⇝ Suppose, to another square in the same column

Each state has $8 \times 7 = 56$ successors

The heuristic cost function h

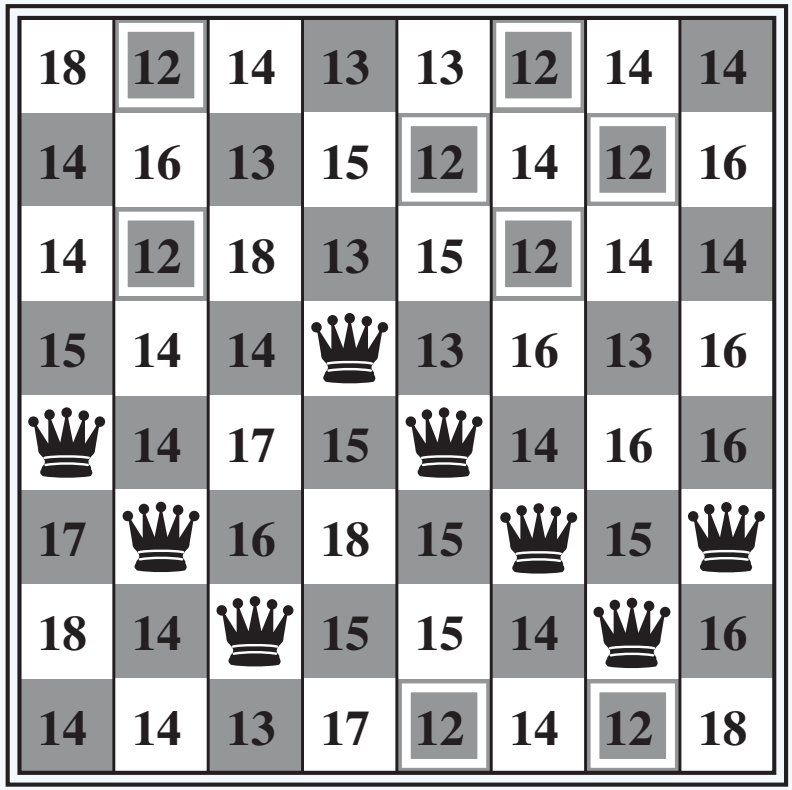
- The number of pairs of queens that are attacking each other
- (either directly or indirectly)

The global minimum of this function is zero

- ⇝ It occurs only at perfect solutions

Hill-climbing search (cont.)

Example



The current state has an estimated heuristic cost $h = 17$

The values of all its successors, if we move a queen in its column

- The best successors have $h = 12$

If more than one best successor, randomly break the tie

Hill-climbing search (cont.)

Hill climbing is also called **greedy local search**

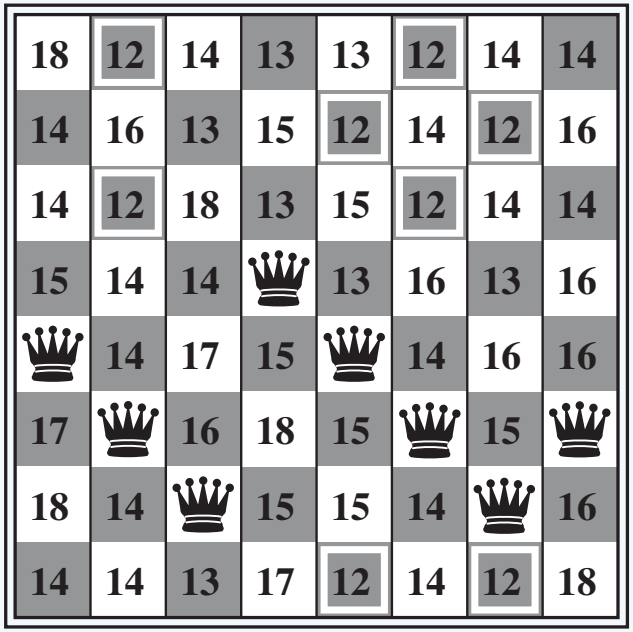
- Pick a good neighbour state without thinking ahead where to go next

Hill climbing often makes rapid progress toward a solution because

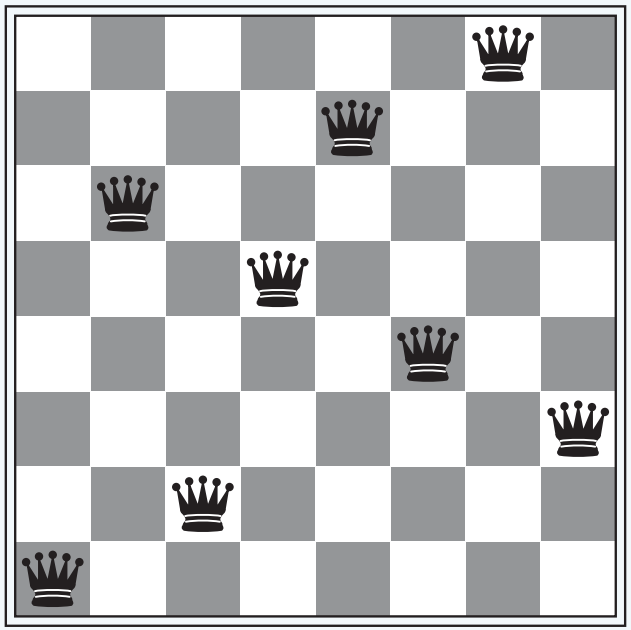
- It is usually quite easy to improve a bad state

Hill-climbing search (cont.)

Example



(a)



(b)

From state (a) it takes five steps to reach state (b)

- State (b) has $h = 1$, it is very nearly a solution
- Every successor of (b) has a higher cost (dead-end)

Hill-climbing search (cont.)

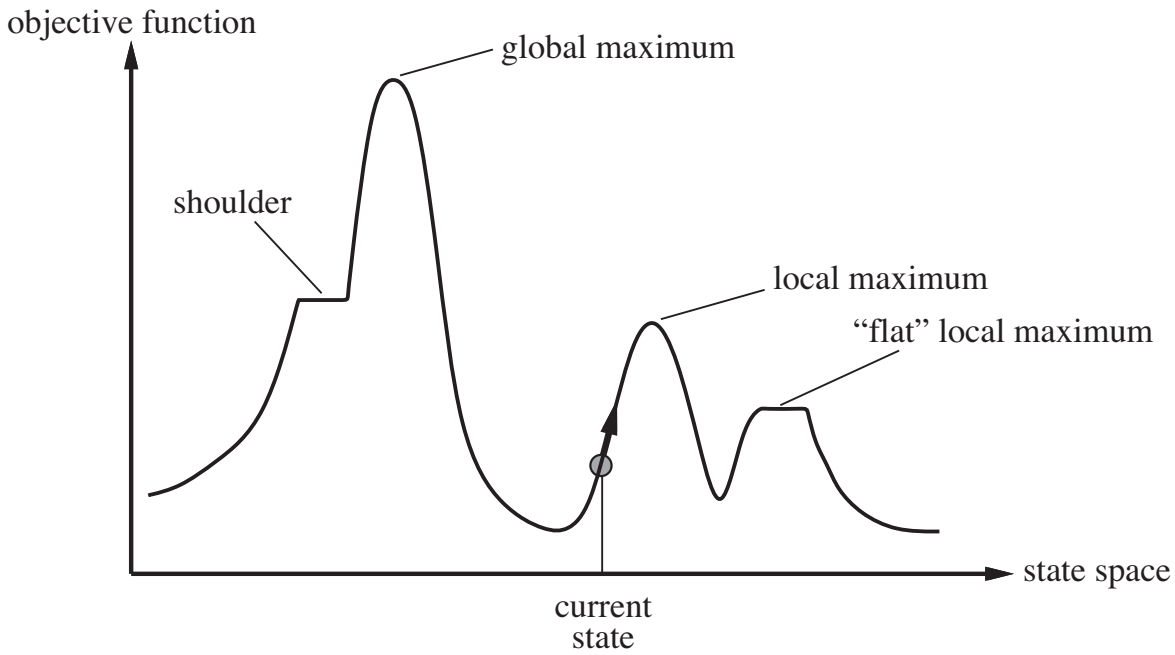
Hill climbing often gets stuck

Local maxima

- A peak that is higher than each of its neighbouring states
- But, lower than global maximum

Plateaux

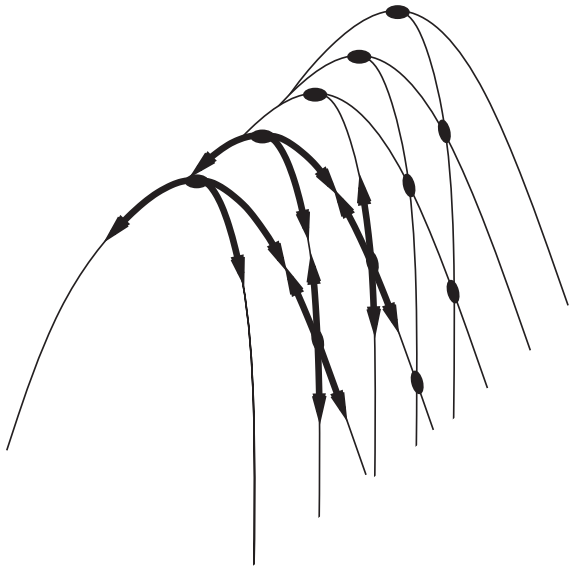
- A flat area of the state-space landscape
- A flat local maximum or a **shoulder**



Hill-climbing search (cont.)

Ridges

- A sequence of local maxima not directly connected to each other



The grid of states (dark circles)

- Shown as superimposed on a ridge

From each local maximum

- All available actions point downhill

In each case, the algorithm reaches a no-more-progress point

Hill-climbing search (cont.)

Example

Steepest-ascent hill climbing gets stuck 86% of the time

- Solves 14% of problem instances

From a random 8-queens state

It is quick, 4 steps on average when it succeeds and 3 when it gets stuck

- Not bad for a state space with $8^8 \approx 17$M states

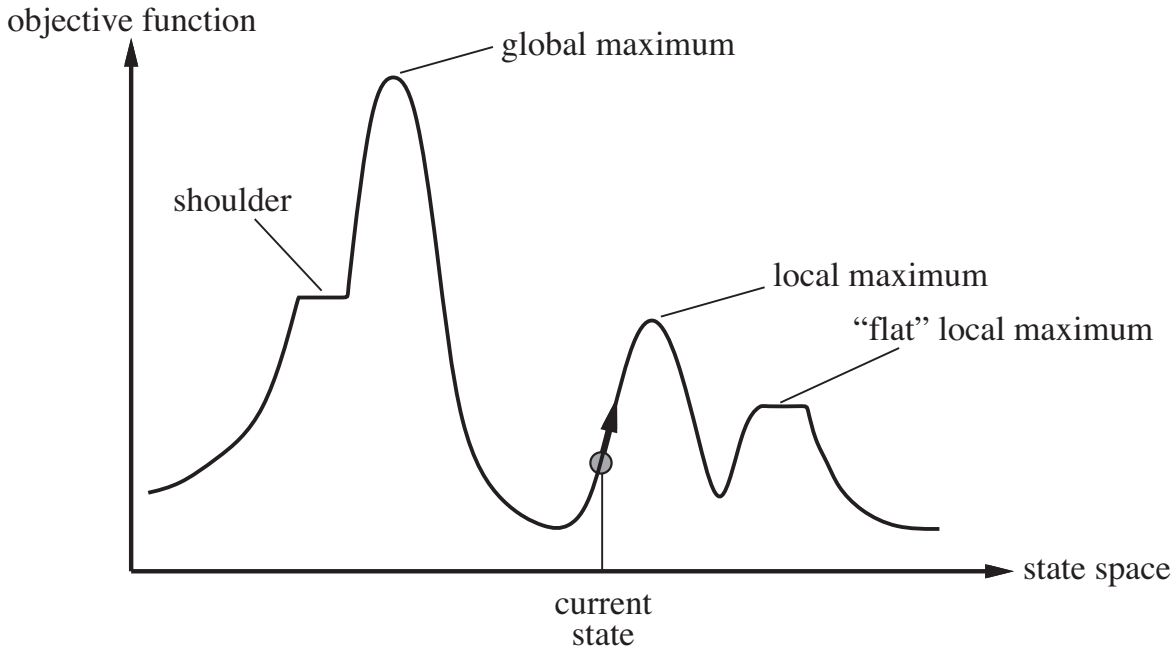
Hill-climbing search (cont.)

Plateaus where best successors have the same value as the current state

⇝ The algorithms halts

It may be good idea to keep going, to allow a **sideways move**

- In the hope that the plateau is really a shoulder



Hill-climbing search (cont.)

Consider always allowing sideways moves when there is no uphill

- An infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder

A solution

$\rightsquigarrow$ Limit the number of consecutive sideways moves allowed

Hill-climbing search (cont.)

Example

A limit of 100 consecutive sideways moves in the 8-queens problem

- Percentage of solved instances by hill climbing raises from 14% to 94%

The algorithm averages 21 steps for successful instance and 64 for failure

Hill-climbing search (cont.)

Many variants of hill climbing have been invented

Stochastic hill climbing chooses at random from the uphill moves, with a probability of selection that can vary with the steepness of the uphill move

- This usually converges more slowly than steepest ascent
- In some state landscapes, it finds better solutions

First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one that is better than current state

- Good strategy, when a state has thousands of successors

Hill-climbing search (cont.)

The hill-climbing algorithms described so far are incomplete

- Even when a goal exist, they can get stuck on local maxima

Random-restart hill climbing conducts a series of hill-climbings

- From randomly generated start states, until a goal is found

It is trivially complete with probability approaching 1

- It will eventually generate a goal state as the initial state

If each hill-climbing search has a probability p of success

$\rightsquigarrow$ Then, the expected number of restarts required is $1/p$

Hill-climbing search (cont.)

Remark

Success of hill climbing depends on the shape of the state-space

- Random-restart hill climbing will find a good solution very quickly
- If there are few local maxima and plateaux

Many real problems have a rather complex state space landscape

- NP-hard problems often have an exponential number of local maxima

Despite this, a reasonably good local maximum can often be found

Simulated annealing

Local search and optimisation

Simulated annealing

Consider a hill-climbing algorithm that never makes 'downhill' moves

- States with lower value will never be visited

The algorithm is guaranteed to be incomplete

- Because it can get stuck on a local maximum

A random walk is complete

- It moves to a successor chosen at random from the set of successors
- But, it is also extremely inefficient

Simulated annealing (cont.)

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

Simulated annealing (cont.)

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow next.\text{VALUE} - current.\text{VALUE}$
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

The innermost loop of simulated annealing is hill climbing alike

- Instead of picking the best move, it picks a random move
- If the move improves the situation, it is always accepted
- If not, the algorithm accepts the move, with some probability

The probability decreases exponentially with the 'badness' of the move

- The amount ΔE by which the evaluation is worsened

Simulated annealing (cont.)

The probability also decreases as T goes down

- 'Bad' moves are more likely to be allowed at the start when T is high
- They become more unlikely as T decreases

The algorithm will find a global optimum with probability approaching 1

- If the schedule lowers T slowly enough

Local beam search
Local search and optimisation

Local beam search

Keeping just one node in memory might seem to be extreme

- The **local beam search** algorithm keeps track of k states

It begins with k randomly generated states

- At each step, all successors of all k states are generated
- If any one is a goal, the algorithm halts
- Otherwise, the algorithm selects the k best successors
- (from the complete list) and repeats

Local beam search (cont.)

Remark

Local beam search with k states resembles k random restarts in parallel

- In fact, the two algorithms are quite different

- In a random-restart search, each search runs independently
- In a local beam search, information is passed among threads

Remark

The algorithm quickly abandons unfruitful searches

$\rightsquigarrow$ It moves to where the most progress is being made

Local beam search (cont.)

In its simplest form, the algorithm can suffer from a lack of diversity

- The states k can quickly become concentrated in a small region

$\rightsquigarrow$ (expensive hill climbing)

A variant called **stochastic beam search** helps with this issue

- Stochastic beam search chooses k successors at random
- it does not pick the best k from the pool of candidates

Probability of choosing a given successor increases with its value

Local beam search (cont.)

Stochastic beam search bears the process of natural selection

‘Successors’ (offspring) of a ‘state’ (organism) populate the next generation

- Selection according to their respective ‘value’ (fitness)

Genetic algorithms

Local search and optimisation

Genetic algorithms

A **genetic algorithm** or **GA** is a variant of stochastic beam search

- Successor states are generated by combining two parent states
- (Rather than by modifying a single state)

Also genetic algorithms begin with a set of k randomly generated states (**population**) and each state (**individual**) is represented as a string

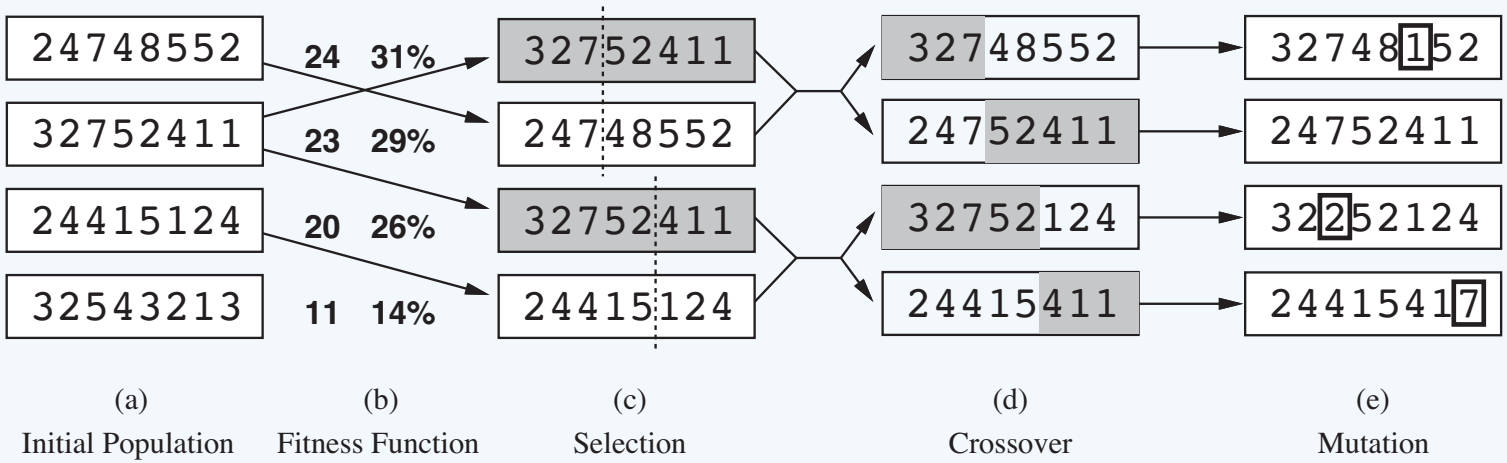
- The string is over a finite alphabet (commonly, a string of `0`s and `1`s)

Genetic algorithms (cont.)

Example

Alternatively, the state could be represented as 8 digits

- Each in the range from 1 to 8

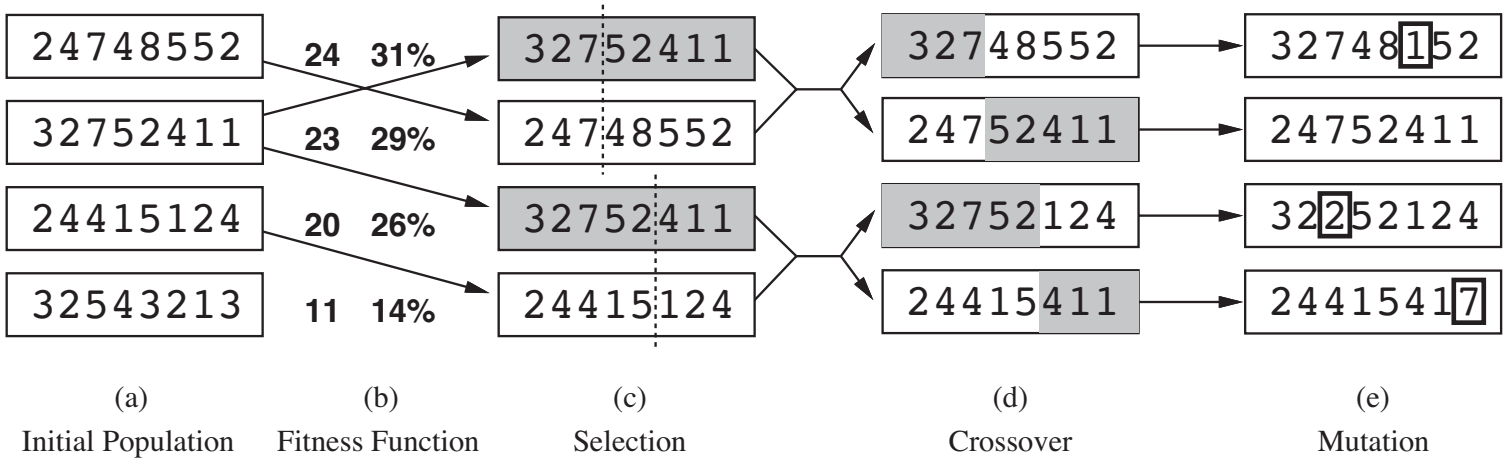


A population of four 8-digit strings representing 8-queens states (a)

- States are ranked

Genetic algorithms (cont.)

The next step is the production of a new generation of states



In (b), each state is rated by the **objective** (**fitness**) **function**

- A fitness function should take higher values for better state

The number of non-attacking pairs of queens (28 for a solution)

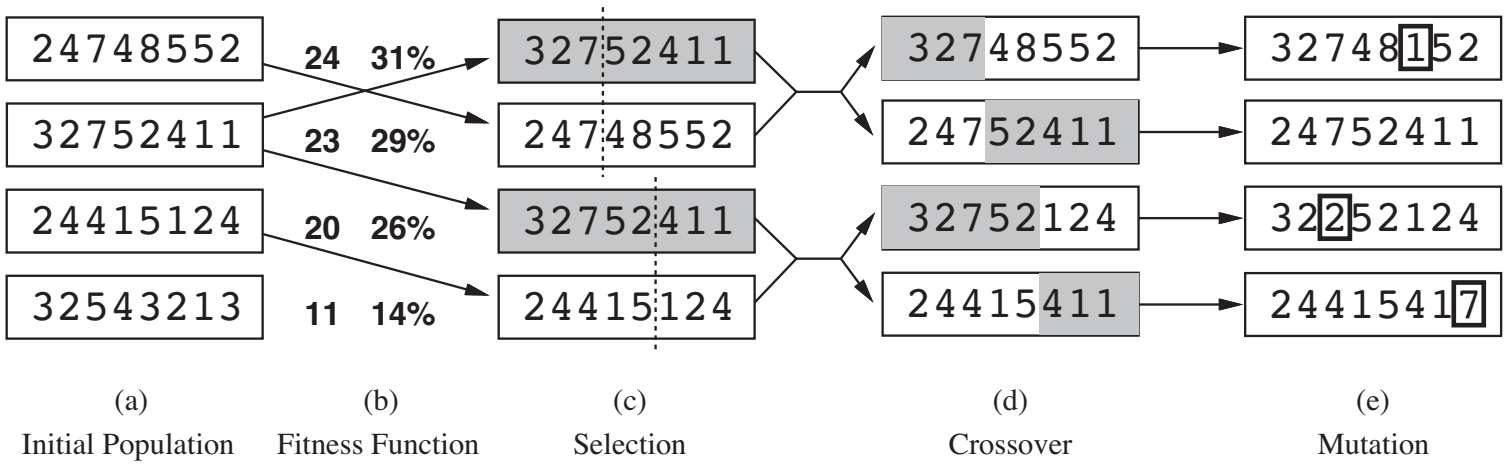
- The values for the four states are 24, 23, 20, and 11

Genetic algorithms (cont.)

Probability of being chosen for reproducing is directly proportional to fitness

This is specific of this variant of the genetic algorithm

Genetic algorithms (cont.)



In (c), two state pairs are selected at random for reproduction

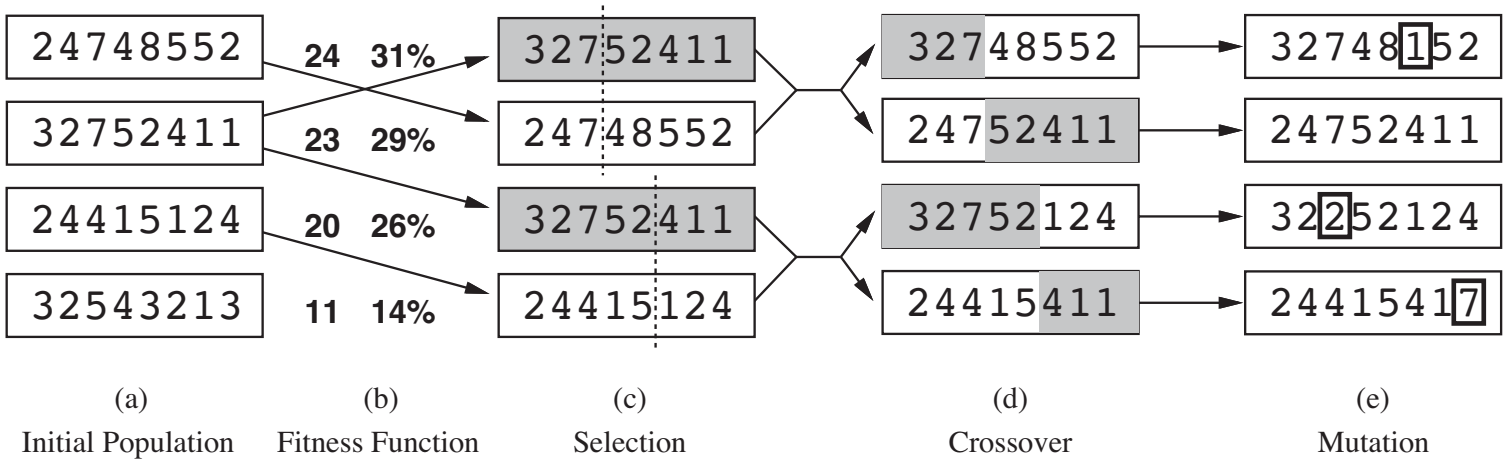
- In accordance with the probabilities in (b)

Notice that one individual is selected twice and one not at all

For each pair to be mated, a **crossover point** is chosen randomly

- After the third digit in the first pair
- After the fifth digit in the second pair

Genetic algorithms (cont.)



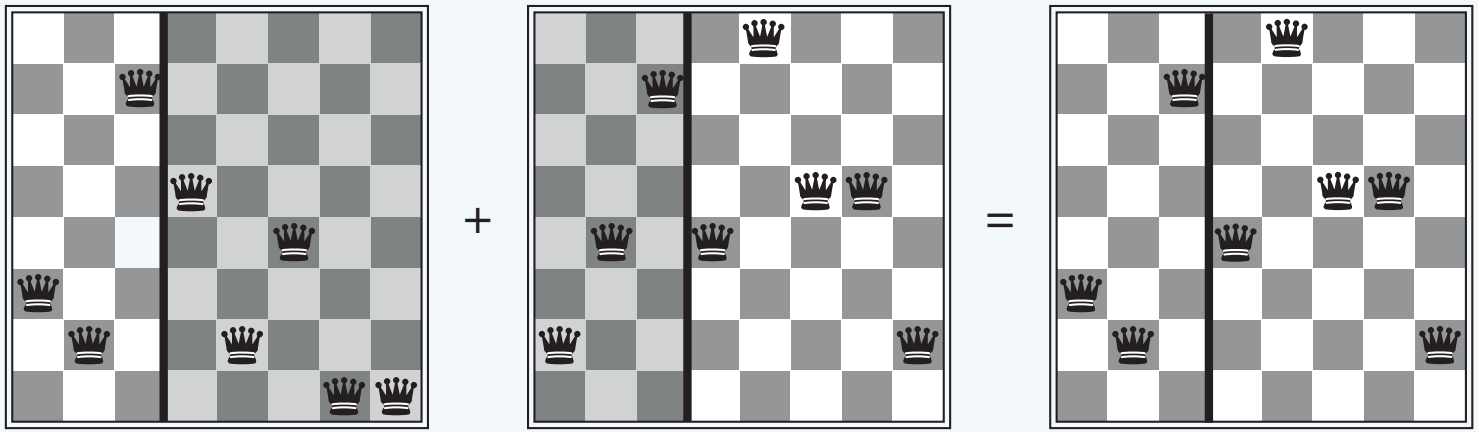
Offspring are created by crossing over parents at crossover point

- First child of first pair gets the first three digits from first parent
- The remaining digits are from from second parent
- Second child gets the first three digits from second parent
- The rest from the first parent

Genetic algorithms (cont.)

Example

The 8-queens states involved in this reproduction step



Crossover can produce a state that is a long way from either parent state

- When two parent states are quite different

Genetic algorithms (cont.)

Common that the population is diverse early on in the process

- Crossover initially takes large steps in the state space
- Steps get smaller later, when individuals get similar

Genetic algorithms (cont.)

function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual
  inputs: *population*, a set of individuals
    FITNESS-FN, 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-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*, according to FITNESS-FN

---

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))

Local search in continuous spaces

Local search in continuous spaces

The discrete/continuous characterisation applies to the environment state

- How time is handled, to the percepts and actions of the agent
- Most real-world environments are continuous

Local search in continuous spaces (cont.)

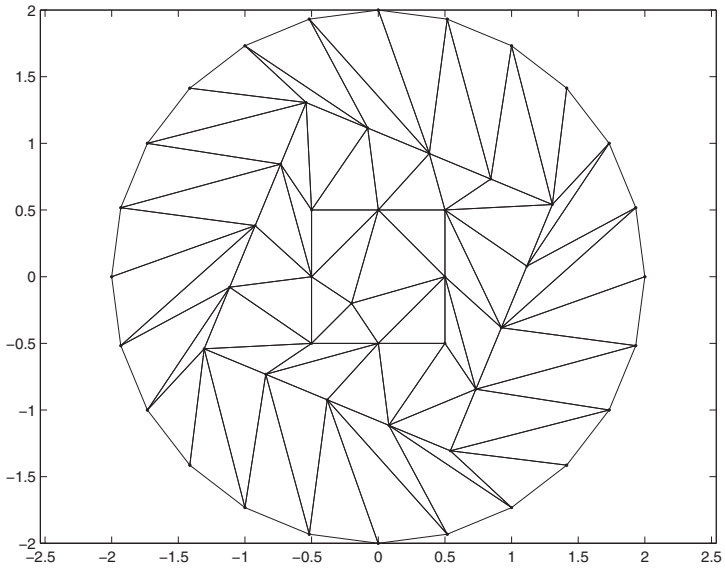
Example

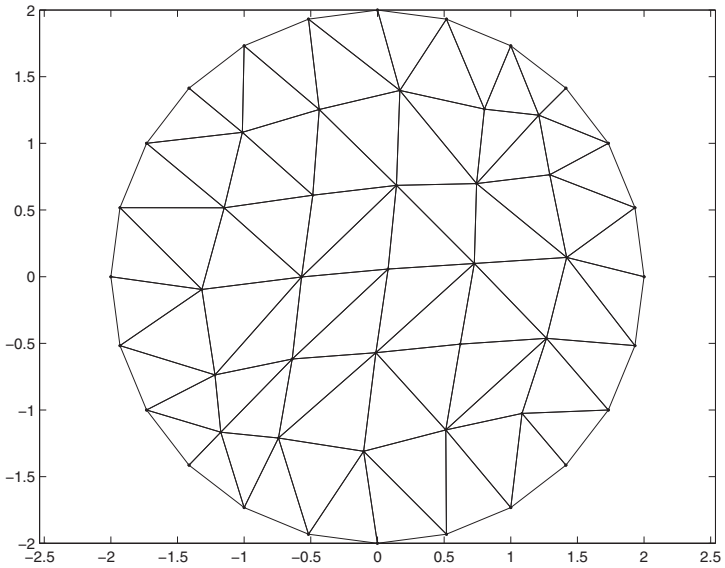
Consider a given triangulation of a domain $\Omega \subset \mathbb{R}^2$

We want to modify the location of the vertices of the triangles inside Ω

- In order to optimise some measure of triangles' quality

$\rightsquigarrow$ Minimise distortion wrt an equilateral triangle





Local search in continuous spaces (cont.)

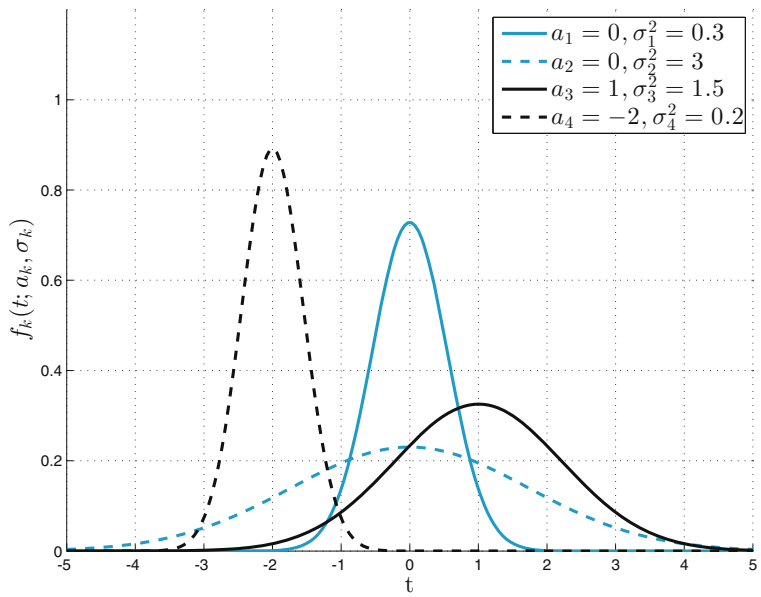
Example

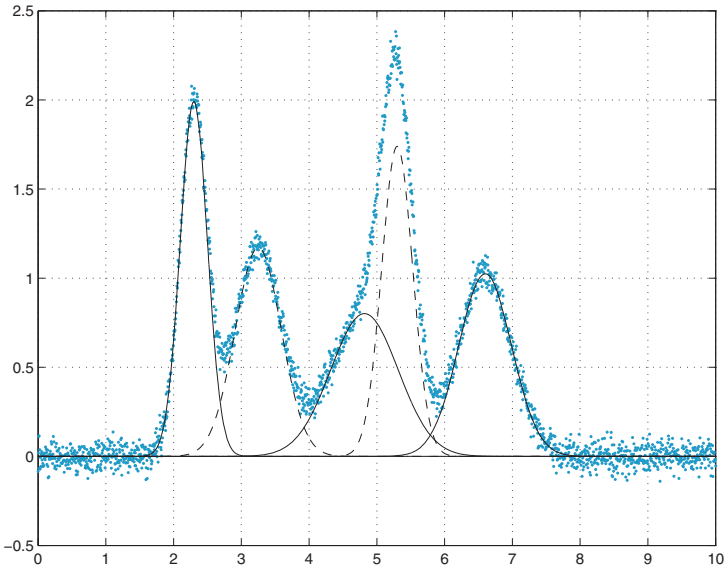
Applications of voice identification, like user authentication on phones

- Compress an acoustic signal into a set of parameters
- Parameters must characterise the signal

The signal intensity is modelled as a sum of m Gaussian functions (peaks)

- Each Gaussian parameterised by two coefficients (centre + spread)





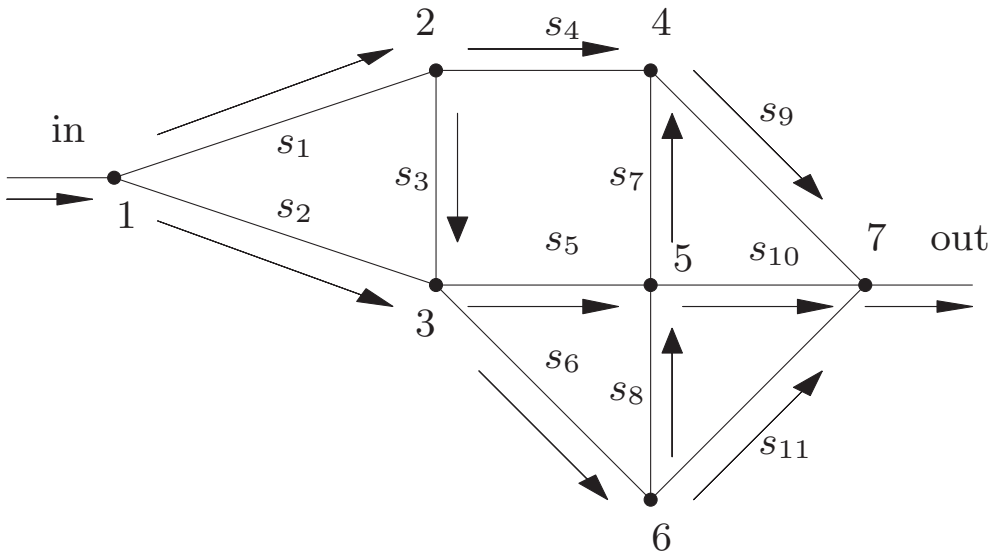
Find the set of $(2 \times m)$ parameters that 'best' represent signal intensity

Local search in continuous spaces (cont.)

Example

Consider a network of n roads and p cross roads

- Every minute M vehicles travel through the network
- On the j-th road the maximum speed limit is $v_{j,m}$ [km min^{-1}]
- Max $\rho_{j,m}$ vehicles per km can transit on the j-th road s_j



Find the density ρ_j [vehicles km^{-1}] on road s_j s.t. $\rho_j \in [0, \rho_{j,m}]$

- The one that minimises average travel time from entrance to exit

Local search in continuous spaces (cont.)

The algorithms we described cannot handle continuous state/action spaces

- At least not in their basic formulation

This is because those problems have infinite branching factors

Local search for finding optimal solutions in continuous spaces

$\rightsquigarrow$ **Numerical optimisation**