Course: Natural Computing

2. Discrete Optimisation Algorithms

Simulated Annealing and Genetic Algorithms

J. Michael Herrmann
School of Informatics, University of Edinburgh
michael.herrmann@ed.ac.uk, +44 131 6 517177

Overview

Simulated Annealing:	Inspired by statistical physics (I)
Genetic Algorithms:	Inspired by natural evolution (II+III)
Ant Colony Optimisation*:	Inspired by reinforcement learning

*Discussion on Friday this week and in the tutorials

I. Simulated Annealing

Course: Natural Computing (week 2)

J. Michael Herrmann
School of Informatics, University of Edinburgh
michael.herrmann@ed.ac.uk, +44 131 6 517177

Simulated Annealing

- Inspired by the traditional annealing process in metals:
    - at higher temperatures the atoms are more random
    - when cooling down, the atoms will locally crystallise
    - during repeated temperature changes the crystals will become the dominant phase of the metal
- First applied to optimisation problems by Kirkpatrick, Gelatt, Vecchi (1983)

Simulated Annealing: Physics background

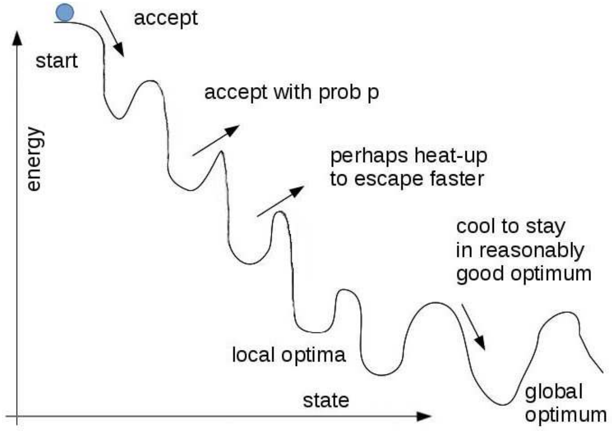
- Find low-energy states of type $x = (x_1, \ldots, x_D)$, $x_i \in \{0, 1\}$
- Describing D stochastic atoms with $x_i = \begin{cases} 1 & \text{in place} \\ 0 & \text{not in place} \end{cases}$
- States occupancy follows Boltzmann distribution

$$p(x) = \frac{\exp\left(-\frac{1}{T} F(x)\right)}{\sum_y \exp\left(-\frac{1}{T} F(y)\right)}$$

  High energy $F(x)$ states are rare, low energy states frequent, and (e.g.) $F((1, \ldots, 1, 1, 1, \ldots, 1)) \leq F((1, \ldots, 1, 0, 1, \ldots, 1))$
  This is counterbalanced by temperature T:
  - If $T \gg 1$, all x will have nearly the same probability
  - If $T \gtrapprox 0$, the x with low energy have highest probability
- I.e. cooling leads to a state with low energy, but not necessarily to the state with the lowest energy.

Simulated Annealing: Intuition towards an algorithm

Increase the chance that a state that is close to the global energy minimum is found by repeated heating and cooling



How and how often to cool-down and heat-up to get good results?

Simulated Annealing: Algorithm

1. Set $t = 1$ (time), $T = 1$, x_0 random, determine $F(x_0)$
2. Generate new solution $x_t = O_k(x_{t-1}) \in X$ from previous solution by randomly chosen operator
3. Determine energy $F(x_t)$
   - if $F(x_t) \leq \Theta$ (or if $t > t_{\max}$), then return(solution)
4. If $t > 0$, determine $\Delta F = F(x_t) - F(x_{t-1})$
   - If $\Delta F < 0$, then continue
   - If $\Delta F \geq 0$, then accept with probability $\sim \exp\left(-\frac{1}{T}\Delta F\right)$,
     otherwise set $x_t = x_{t-1}$
5. $t{+}{+}$ and change T in a suitable way
6. Goto 2

for additional information, see next page

Simulated Annealing: Remarks

- The initial state x_0 can contain background information
- The energy F evaluates the solution (cost), such as
  - tour length in a TSP
  - total mismatch in an allocation problem
  - number of violated clauses in a SAT problem
- Operators can be anything, e.g. an operator O_k that flips the k-th bit (for other choices, see below)
- Probability: Revise the initialisation of T, if ΔF is not small.
- Threshold Θ defines when the problem is considered as solved
- Change temperature so that fewer and fewer less good solutions are accepted. This does not have to be done monotonously (see next slide).

Simulated Annealing: How to change temperature?

- If $T(t) = \frac{T_0}{\log(1+t)}$ then an optimal solution is found with probability 1 after sufficiently long time (Johnson, Aragon, McGeoch, Schevon, 1989)
- If temperature is reduced quicker than this, then the algorithm may get stuck in a local minimum and may not find a global optimum ever.
- There are non-monotonous and adaptive (dependent on improvement) schemes
- The *fast annealing* scheme (Szu & Hartley, 1987) which uses a Cauchy distribution instead of a Boltzmann distribution.

How to create new solutions?

What operators to use for the generation of new solutions?

New solutions should be selected from a certain neighbourhood

- single flips as above (i.e. random change of one component)
- flip small random subset of components (or within an ε-ball)
- temperature-related (wider jumps and higher acceptance)
- adaptive in order to "learn" directions in space that promise high improvements
  - remember components of x that led to an improvement
  - remember correlations among the components

Practically, start with simplest operators, but also try to derive opertors as implied by the problem

Simulated Annealing: Conclusions

- General scheme: $(\textit{Mutate, Select})_{\text{repeat}}$ and *Check Parameters*
- Usually only a single agent: Population size here $N = 1$
- Simple algorithm, easy to set up, physics background is not required to run the algorithm
- May be faster than purely random search, in particular if the cooling scheme and the neighbourhood width are appropriately chosen (or adapted)
- Note that physical systems may not be in their minimal-energy configuration even at low temperature, i.e. local optimal also occur in the real world, see e.g. Prinz Rupert's drops

II. Genetic Algorithms

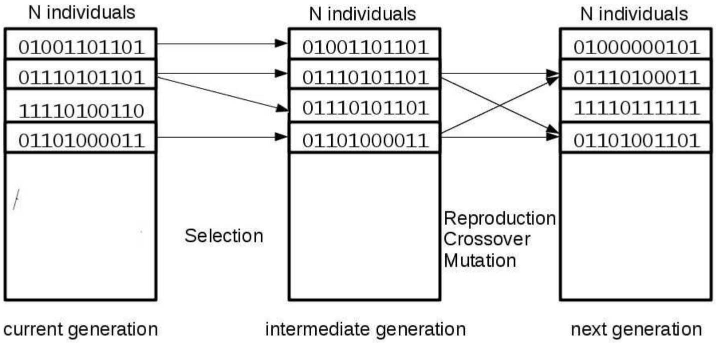
Course: Natural Computing (week 2)

J. Michael Herrmann
School of Informatics, University of Edinburgh
michael.herrmann@ed.ac.uk, +44 131 6 517177

Genetic algorithms

- Natural evolution as an inspiration for problem solving
- E.g. Flight: requires wings, a visual system, air-flow sensors, smart control, efficient energy supply, light-weight body, ...
- Encoding of candidate solutions in form of strings of numbers (genotype)
- Decoding of (usually binary) genotype into phenotype (discrete or continuous) and evaluation of phenotype by fitness function
- Operators: mutation, recombinations (on genotypes)
- Selection and generation of off-spring

Genetic Algorithms: How it works



Cyclic: Irrelevant whether selection first or reproduction first

Intermediate population contains identical copies

Usually large populations (e.g. $N = 1000$)

Problem Solving by Genetic Algorithms

- Find a representation of solutions
  e.g., a description of a robot [expressed as a bit string]
  $x =$ (lengths of legs, angle limits, motor power, ...)
- Define an objective function $\rightarrow$ fitness function (to be maximised) that can be calculated for each x
- Generate a population of candidate solutions (bag of strings)
- Evolution scheme for the solutions:
  - Evaluate the candidate solutions
  - Choose (preferentially) high-fitness solutions
  - Modify some of them
  - Start again unless a termination criterion is met

A Simple Example
The Tutor Allocation Problem (TAP)

Jobs: Job_1, Job_2,... Job_m
Job_i is a single tutorial to be taught

- subject
  e.g. NAT, Introduction to Java Programming, MLP
- slot
  e.g. Wednesday 4:10-5pm
- place
  e.g. AT 2.12
- knowledge, skills required
  e.g. strong at Java, some knowledge of AI techniques

One tutor teaches each tutorial.
We have a pool of tutors to choose from:
Tutor_A, Tutor_B, ...,Tutor_X
Properties of tutors

- knowledge/skills
- cost per hour
- time preferences
- location preferences
- optimal number of jobs
- preference for connected time slots

A Simple Example

The Tutor Allocation Problem (TAP)

Solutions: A solution is an allocation of tutors to jobs, e.g.

Job	1	2	3	4	5	6	7	8	9	10
Tutor	A	B	C	D	E	F	G	H	I	J

Each job-tutor pairing can be given a score, based on how good the knowledge/skill match is, e.g.

- Tutor A: some C++, strong at AI
- Job 1: strong Java, some AI useful

➡ a reasonable match, though not perfect

A function $f_s\,(\text{job}, \text{tutor})$ calculates a numerical score for any pairing

A Simple Example
The Tutor Allocation Problem (TAP)

The whole solution can be given a score, based on

- sum of scores for job-tutor pairings
- total cost of solution
- hard constraints
- tutor preferences

The total score will be calculated from the scores for the individual parts.

The **problem** is to find the solution with the best score.

A Simple Example
Possible Methods

Exhaustive search?

- 5 tutors, 10 jobs $= 9.8 \times 10^6$ solutions
- 10 tutors, 20 jobs $= 1.0 \times 10^{20}$ solutions
- 15 tutors, 30 jobs $= 1.92 \times 10^{35}$ solutions
- ...

A Simple Example

Possible Methods

Greedy search?

Job$_1$ – find best tutor for this job

Job$_2$ – find tutor to give the best combined score with the choice for Job$_1$

Job$_3$ – etc.

(or v.v., i.e. starting with Tutor$_A$)

$\Rightarrow$ Almost certain to be sub-optimal since it commits to choices too early

A Simple Example

Possible Methods

Hill-climbing local search (highest improvement variant)

Solution$_t$:

Job	1	2	3	4	5	6	7	8	9	10
Tutor	A	B	E	A	B	B	$\boxed{\text{D}}$	C	E	D

Suppose $\boxed{\text{D}}$ is the worst scorer. Try A, B, C, E

Solution$_{t+1}$:

Job	1	2	3	4	5	6	7	8	9	10
Tutor	A	B	E	A	B	B	$\boxed{\text{A}}$	C	E	D

Continue until no improvement possible.

Prone to local maxima.

Genetic Algorithms

How about trying a biologically inspired solution based on genetics?

1. Generate a population of solutions

Generation$_t$:

Job	1	2	3	4	5	6	7	8	9	10
Solution$_1$	A	B	E	A	B	B	A	C	E	D
Solution$_2$	E	C	B	A	D	E	C	D	A	D
$\vdots$										
Solution$_N$	C	E	E	D	D	A	B	A	C	A

2. Give each solution a score, called fitness.
3. Create a new generation of solution by
   1. selecting fit solutions
   2. breeding new solutions from old ones and add to generation$_{t+1}$
4. When a sufficiently good solution has been found, stop.

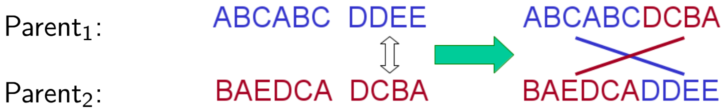
A Simple Genetic Algorithm

- Selection (out of n solutions, greedy type):
  - Calculate $\sum_i f_s\,(\text{Job}_i, \text{Tutor}_i)$ for each solution S
  - Rank solutions
  - Choose the k best scorers ($1 \leq k \leq N$)
- Breeding (Mixing good solutions):
  - take a few of the good solutions as parents
  - cut in halves, cross, and re-glue (see next slide)
- Mutation:
  - generate copies of the mixed solutions with very few modifications
  - e.g. for $k = N/2$: two “children” for each of them

Recombination and Mutation

How does breeding work?

1. Reproduction:
   Copy solution$_i$ unchanged into the next generation
2. Crossover (here: single cut):



Exchange of genetic material to from children

Recombination and Mutation

How does breeding work?

3. Mutation

- Change one value in a solution to a random new value:

AEBCABDDCE → AEBCABDACE

- Swap two values

AEBDABDCCE → AEBCABDDCE

- Lots of others!

Mutation is usually done after reproduction/crossover with low probability (e.g. 1%)

Intermediate conclusion

Together selection, mutation, and crossover can be creative:

- Selection + Mutation = Continual improvement
- Selection + Recombination = Innovation

- The algorithm without recombination, i.e. only mutation and selection, is known as *Evolution Strategy* (H.-P. Schwefel, 1960s)
- Mutation is similar as in simulated annealing
- Crossover can combine the best parts of present solutions
  - can lead to large jumps in the search space
  - keeps groups of good components together
  - recombines these groups in new ways
- Representation is critical, as are the rates p_c and p_m
- Works best for smooth fitness function and diverse population

Outlook to a theory of GA

- How does it work? The schema theorem* gives some explanation
    - What is a schema? A non-empty subset of a string (in other words: a string with some wildcards)
    - Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generation of a genetic algorithm.
- Many variants of GA: E.g. for multi-objective problems
- Performance analysis: Benchmarks, applications
- General formulation, theory, convergence etc.

Genetic Algorithms
III. The canonical GA

Course: Natural Computing (week 2)

J. Michael Herrmann
School of Informatics, University of Edinburgh
michael.herrmann@ed.ac.uk, +44 131 6 517177

The Canonical Genetic Algorithm: Conventions

1. Old population
2. Selection
3. Intermediate population
4. Recombination
5. Mutation
1. New population

$\Bigg\}$ one generation

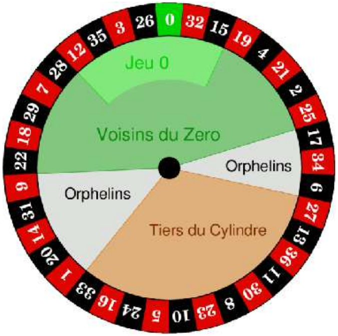
- A population is a (multi-) set of individuals
- An individual (genotype, chromosome) is a string $S \in \mathcal{A}^L$
  ($\mathcal{A}$: alphabet, often: $\mathcal{A} = \{0, 1\}$)
- Fitness = objective function = evaluation function.
  Fitness values are often replaced by ranks (high to low)

Roulette Wheel Selection

I. Plain variant

Mean fitness $\bar{f} = \frac{1}{n}\Sigma_i f_i$ $\Longrightarrow$ Normalized fitness: $\frac{f_i}{\bar{f}}$
(from now on short: fitness)

- Each time the ball spins one individual is selected for the intermediate population
- Stochastic sampling with replacement
- Ratio of fitness to average fitness determines number of offspring, i.e. a new individual is a copy of an old individual (of fitness f_i) with probability $\frac{f_i}{n\bar{f}}$
- If $f_i = \bar{f}$ then the individual "survives" with probability $1 - \left(1 - \frac{1}{n}\right)^n$



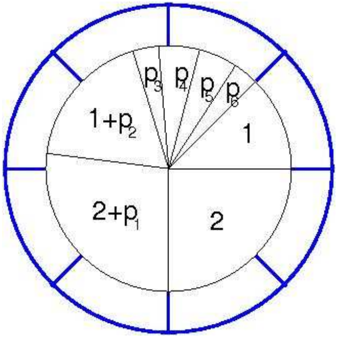
Sector (French) bets in roulette: Here, the size of the sector represents the relative fitness of an individual

Roulette Wheel Selection

II. A more practical variant

Mean fitness $\bar{f} = \frac{1}{n}\Sigma_i f_i \quad \Longrightarrow \quad$ Normalized fitness: $\frac{f_i}{\bar{f}}$

- Remainder stochastic sampling
- Ratio of fitness to average fitness determines number of offspring
- If $f_i = \bar{f}$: the individual survives
- If $f_i < \bar{f}$: survives with prob. $\frac{f_i}{\bar{f}}$
- If $f_i > \bar{f}$: number of offspring $\text{int}\left(\frac{f_i}{\bar{f}}\right)$ and possibly one more with probability $\frac{f_i}{\bar{f}} - \text{int}\left(\frac{f_i}{\bar{f}}\right)$



Now: Only the outer wheel with equidistant pointers spins once and pointers in each sector are counted

Both variants are equivalent in the sense that they produce an unbiased sample of the fitness in the population, i.e. a new individual is a copy of an old individual (of fitness f_i) with probability $\frac{f_i}{n\bar{f}}$

From intermediate to new population

Preparation:

- Population was already shuffled by selection (but may contain multiple copies of the same string)
- Individuals are strings of equal length L
- Choose a probability p_c:

Crossover:

- Choose a pair of individuals
- With probability p_c:
  - choose a position from 1 to $L - 1$
  - cut both individuals after this position
  - re-attach crossed: xyxxxyyy, abbabbab $\rightarrow$ xyxxbbab, abbaxyyy
- Move the obtained pair to the new population (even if not crossed over)
- Repeat for the remaining pairs (assert n even)

From intermediate to new population

Preparation:

- Crossover finished
- Individuals are strings of length L made from k different char's
- Choose a (small) probability p_m (possibly rank-dependent)

Mutation:

- For all individuals (from new population)
    - for each position from 1 to L
    - with probability p_m:
    - set the character (bit if binary) to a random value or change it [this gives $\frac{k}{k-1}$ (i.e. twice if binary) the effect! Canonical: binary, switch]
- The obtained mutants (possibly including some unmutated individuals) form the new population

The canonical GA: Summary

- Evaluation function F (raw fitness) gives a score $F(i) = f_i$ to each individual solution $i \in \{1, \ldots, n\}$
- If $\bar{f}$ is the average evaluation over the whole population of n individuals then the **fitness** of i is $f_i/\bar{f}$
- Probability of **selection** of a solution with evaluation f_i is $f_i/\sum_i f_i$
- Select two parents at random from the intermediate population. Apply crossover with probability p_c, with probability $1 - p_c$ copy the parents unchanged into the next generation — reproduction. Typical value: $p_c = 0.7$
  Crossover: from the 2 parents create 2 children using 1-point crossover. Select crossover point uniform-randomly
- **Mutation**: Take each bit in turn and flip it with probability $p_m(1 \to 0$ or $0 \to 1)$. $p_m < 0.01$ usually. Note that the probability p_m is applied differently from p_c
- This is one generation. Repeat for many generations until a **termination** criterion is met.

Simple Example: (bit-wise) “All-Ones”

- Start with a string or zeros (or with a random bit string)
- Fitness is the number of 1s (the “All-Ones” string has highest fitness)
- Selection proportional to fitness
- Operators: Mutations (flip any bit) and Crossover
- How long does it take to find this optimum?
- Details in Tutorial 1

Lessons from the simple examples (see Tutorial 1)

- Mutations are important!
- Large populations, may have high redundancy (many identical or very similar individuals)
  - if they are fit some of them will survive
  - can be useful for exploitation, but not for exploration
- Larger populations may improve exploration, if they are diverse
- Low evolutionary pressure can be helpful: The algorithm will typically find some bits first, (e.g.) 111100 has higher fitness than 000011, but they can be combined into the optimal solution, however, only if some of the latter individuals survive!
- Efficiency by parameter choice ("evolutionary window") speeds up the algorithm and is critical in non-stationary problems.
- Termination can be a non-trivial problem

How well does this work?

- Global search heuristics: Find exact or approximate solutions to optimization problems
- Small problems: Optimal solutions
- Larger problems: optimal or near optimal given enough time
- "Anytime" behaviour of the algorithm: The best so far solution can be used if good enough
- Runs well on parallel machines
- Adding new constraints is easy: Edit or penalise
- Used in a multitude of real applications in many fields such as Bioinformatics, computational science, engineering, robotics, economics, chemistry, manufacturing, mathematics, physics, ...

The reMAINNing Issues

- How do I represent (encode) a solution?
- How should I rate solutions for fitness?
- What initialisation is preferable?
- How large should the population of solutions be?
- How much selection pressure should I apply?
- What form of crossover should I use?
- What form of mutation should I use?
- Should the population be structured (“islands”)?
- Should any form of *elitism* be used?
- Termination should not be forgotten
- Is there anything in GA I can be sure about (e.g. convergence)?

These are examples for questions to be discussed in the Q&A sessions

*Bonus slide: Darwin and beyond

The main ideas in *On the Origin of Species* (1859) have influenced many other areas:

- Theory of natural evolution: Genetics, genomics, bioinformatics
- Evolution of individual learning abilities (Baldwin, 1896)
- Artificial immune systems, resistances in viruses and bacteria
- Evolution of ideas, e.g. in literature (Stanislaw Lem: The Philosophy of Chance, 1968)
- Memetics (R. Dawkins: The Selfish Gene, 1976)
- Neural Darwinism (Gerald Edelman: The Theory of Neuronal Group Selection, 1975 & 1989; less convincing, but inspiring!)
- Neuroevolution (later!)
- Computational finance, markets, agents (later, but just a bit!)