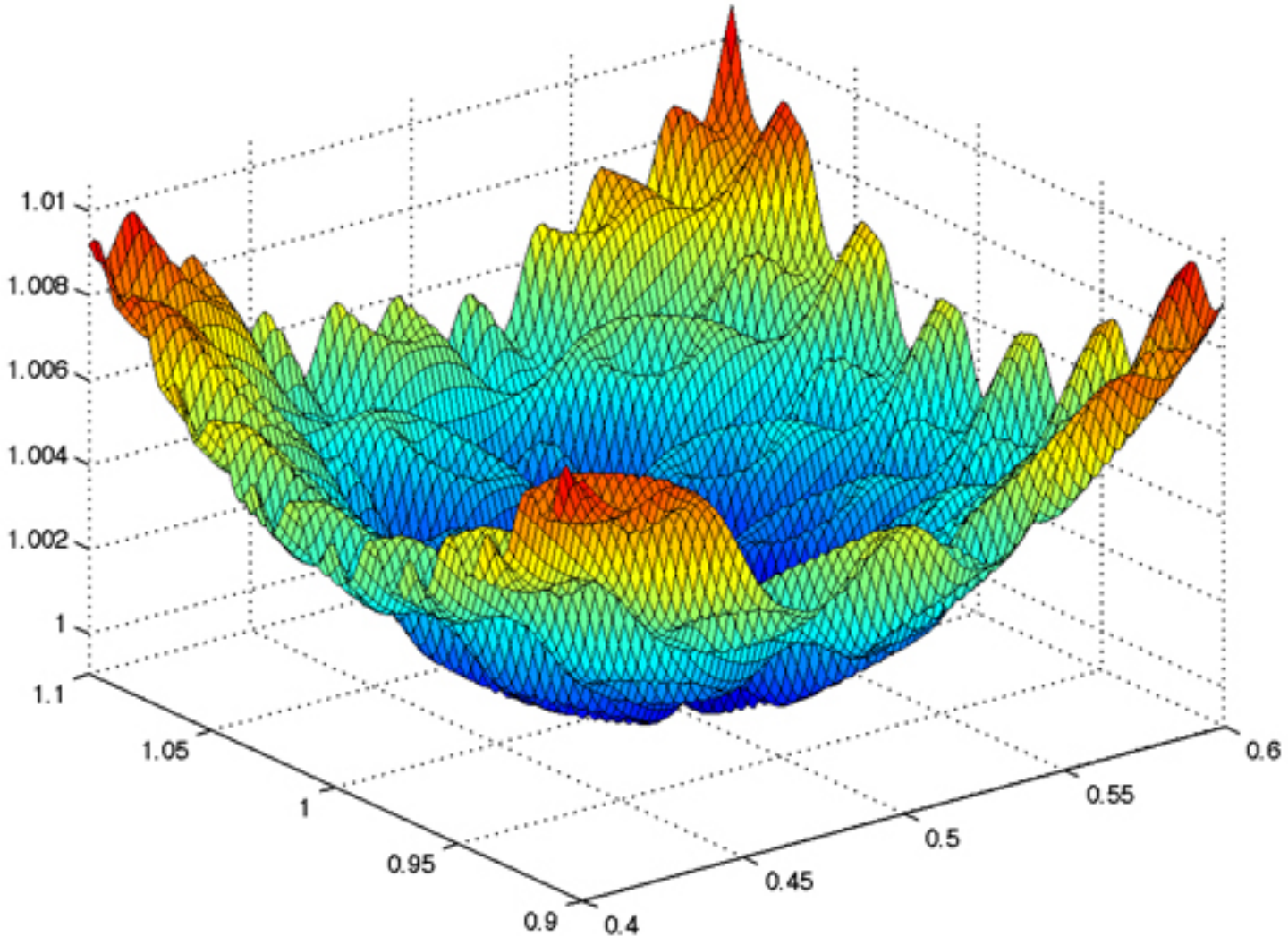
Optimization Algorithms Handbook



Yevette Garrett

First Edition, 2012

ISBN 978-81-323-3642-6

Published by:
University Publications
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1 - Simulated Annealing

Chapter 2 - Ant Colony Optimization

Chapter 3 - Stochastic Optimization

Chapter 4 - Stochastic Programming & CMA-ES

Chapter 5 - Particle Swarm Optimization

Chapter 6 - Bees and Firefly Algorithm

Chapter 7 - Other Concepts in Optimization algorithms

Chapter- 1

Simulated Annealing

Simulated annealing (SA) is a generic probabilistic metaheuristic for the global optimization problem of applied mathematics, namely locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities). For certain problems, simulated annealing may be more effective than exhaustive enumeration — provided that the goal is merely to find an acceptably good solution in a fixed amount of time, rather than the best possible solution.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends both on the difference between the corresponding function values and also on a global parameter *T* (called the *temperature*), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when *T* is large, but increasingly "downhill" as *T* goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local optima—which are the bane of greedier methods.

The method was independently described by Scott Kirkpatrick, C. Daniel Gelatt and Mario P. Vecchi in 1983, and by Vlado Černý in 1985. The method is an adaptation of the Metropolis-Hastings algorithm, a Monte Carlo method to generate sample states of a thermodynamic system, invented by N. Metropolis et al. in 1953.

Overview

In the simulated annealing (SA) method, each point s of the search space is analogous to a state of some physical system, and the function $E(s)$ to be minimized is analogous to the internal energy of the system in that state. The goal is to bring the system, from an arbitrary *initial state*, to a state with the minimum possible energy.

The basic iteration

At each step, the SA heuristic considers some neighbouring state s' of the current state s, and probabilistically decides between moving the system to state s' or staying in state s. These probabilities ultimately lead the system to move to states of lower energy. Typically this step is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhausted.

The neighbours of a state

The neighbours of a state are new states of the problem that are produced after altering the given state in some particular way. For example, in the traveling salesman problem, each state is typically defined as a particular permutation of the cities to be visited. The neighbours of some particular permutation are the permutations that are produced for example by interchanging a pair of adjacent cities. The action taken to alter the solution in order to find neighbouring solutions is called "move" and different "moves" give different neighbours. These moves usually result in minimal alterations of the solution, as the previous example depicts, in order to help an algorithm to optimize the solution to the maximum extent and also to retain the already optimum parts of the solution and affect only the suboptimum parts. In the previous example, the parts of the solution are the parts of the tour.

Searching for neighbours to a state is fundamental to optimization because the final solution will come after a tour of successive neighbours. Simple heuristics move by finding best neighbour after best neighbour and stop when they have reached a solution which has no neighbours that are better solutions. The problem with this approach is that a solution that does not have any immediate neighbours that are better solution is not necessarily the optimum. It would be the optimum if it was shown that *any* kind of alteration of the solution does not give a better solution and not just a particular kind of alteration. For this reason it is said that simple heuristics can only reach **local optima** and not the **global optimum**. Metaheuristics, although they also optimize through the neighbourhood approach, differ from heuristics in that they can move through neighbours that are worse solutions than the current solution. Simulated Annealing in particular doesn't even try to find the best neighbour. The reason for this is that the search can no longer stop in a local optimum and in theory, if the metaheuristic can run for an infinite amount of time, the global optimum will be found.

Acceptance probabilities

The probability of making the transition from the current state s to a candidate new state s' is specified by an *acceptance probability function* $P(e,e',T)$, that depends on the energies $e = E(s)$ and $e' = E(s')$ of the two states, and on a global time-varying parameter T called the *temperature*.

One essential requirement for the probability function P is that it must be nonzero when $e' > e$, meaning that the system may move to the new state even when it is *worse* (has a higher energy) than the current one. It is this feature that prevents the method from becoming stuck in a *local minimum*—a state that is worse than the global minimum, yet better than any of its neighbours.

On the other hand, when T goes to zero, the probability $P(e,e',T)$ must tend to zero if $e' > e$, and to a positive value if $e' < e$. That way, for sufficiently small values of T, the system will increasingly favor moves that go "downhill" (to lower energy values), and avoid those that go "uphill". In particular, when T becomes 0, the procedure will reduce to the greedy algorithm—which makes the move only if it goes downhill.

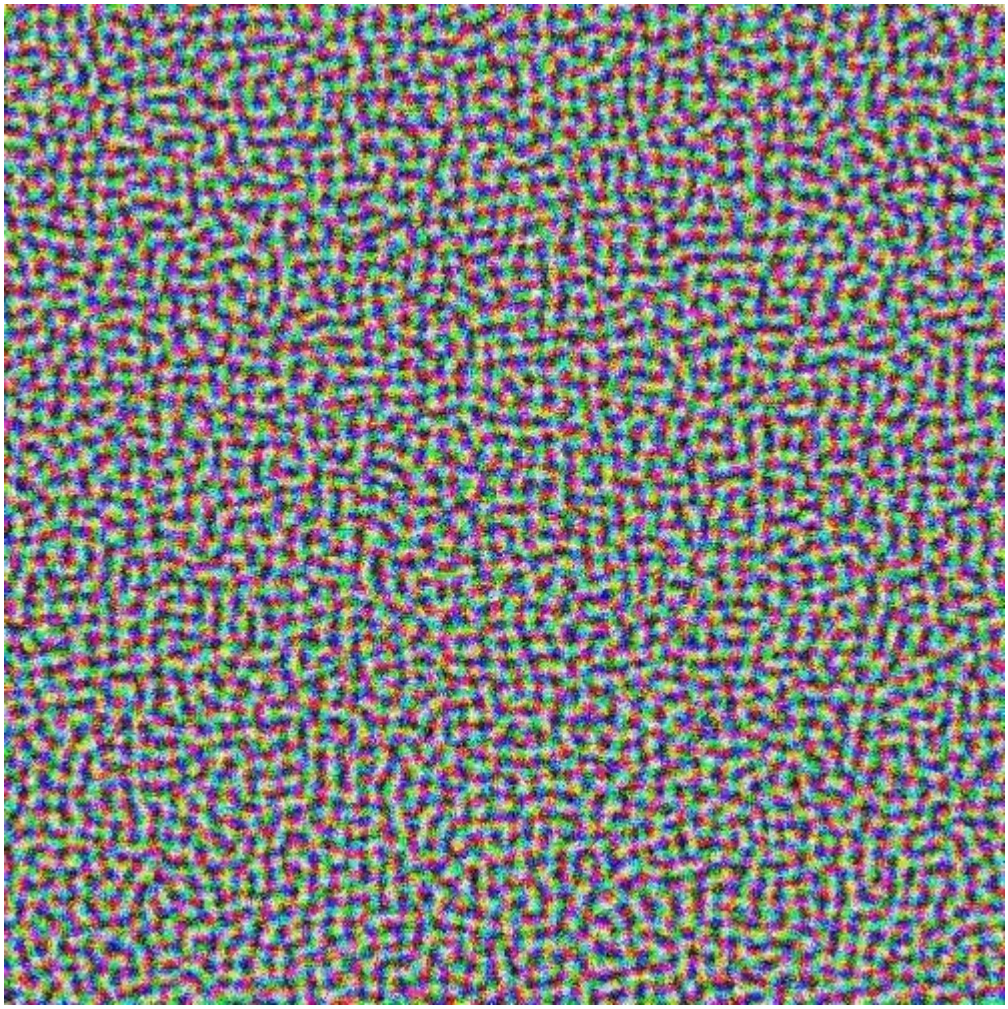
In the original description of SA, the probability $P(e,e',T)$ was defined as 1 when $e' < e$ — i.e., the procedure always moved downhill when it found a way to do so, irrespective of the temperature. Many descriptions and implementations of SA still take this condition as part of the method's definition. However, this condition is not essential for the method to work, and one may argue that it is both counterproductive and contrary to its principle.

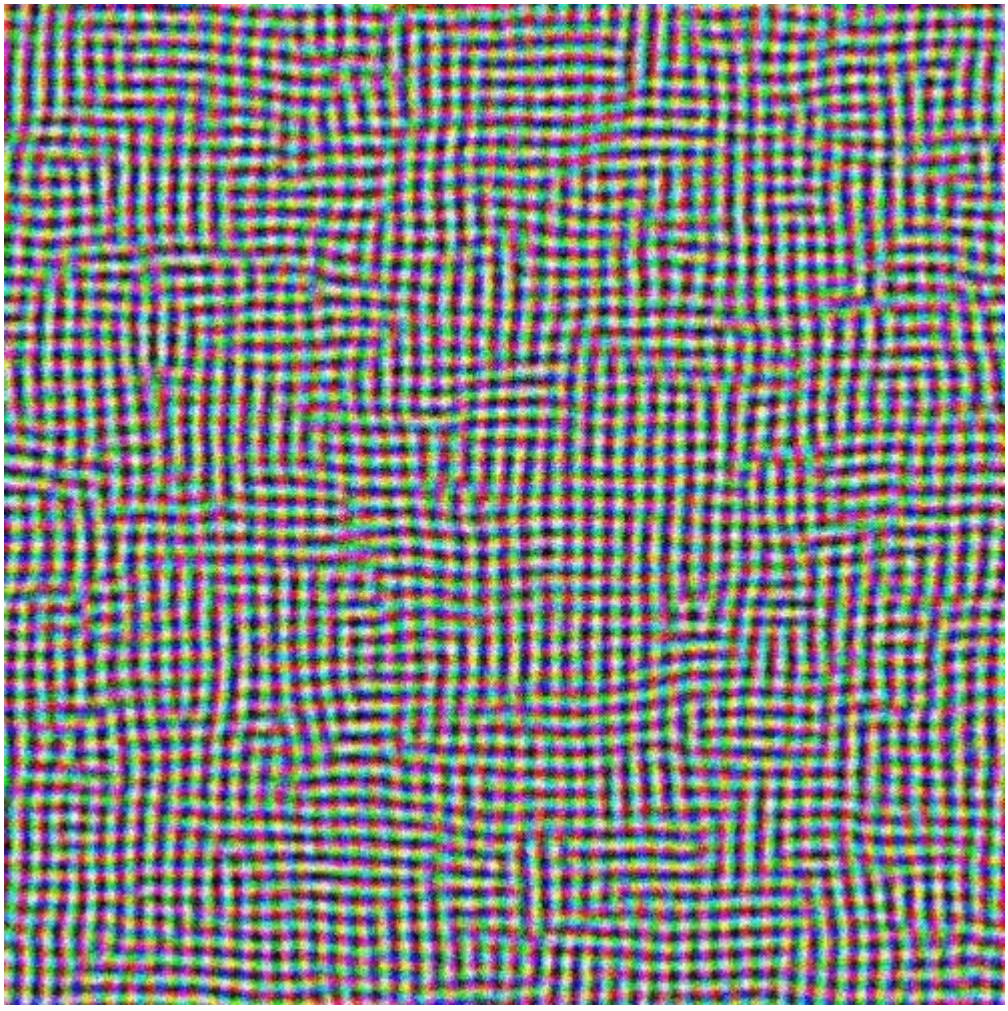
The P function is usually chosen so that the probability of accepting a move decreases when the difference $e' - e$ increases—that is, small uphill moves are more likely than large ones. However, this requirement is not strictly necessary, provided that the above requirements are met.

Given these properties, the temperature T plays a crucial role in controlling the evolution of the state s of the system vis-a-vis its sensitivity to the variations of system energies. To be precise, for a large T, the evolution of s is sensitive to coarser energy variations, while it is sensitive to finer energy variations when T is small.

The annealing schedule

The name and inspiration of the algorithm demand an interesting feature related to the temperature variation to be embedded in the operational characteristics of the algorithm. This necessitates a gradual reduction of the temperature as the simulation proceeds. The algorithm starts initially with T set to a high value (or infinity), and then it is decreased at each step following some *annealing schedule*—which may be specified by the user, but must end with $T = 0$ towards the end of the allotted time budget. In this way, the system is expected to wander initially towards a broad region of the search space containing good solutions, ignoring small features of the energy function; then drift towards low-energy regions that become narrower and narrower; and finally move downhill according to the steepest descent heuristic.





Example illustrating the effect of cooling schedule on the performance of simulated annealing. The problem is to rearrange the pixels of an image so as to minimize a certain potential energy function, which causes similar colours to attract at short range and repel at a slightly larger distance. The elementary moves swap two adjacent pixels. These images were obtained with a fast cooling schedule (left) and a slow cooling schedule (right), producing results similar to amorphous and crystalline solids, respectively.

It can be shown that for any given finite problem, the probability that the simulated annealing algorithm terminates with the global optimal solution approaches 1 as the annealing schedule is extended. This theoretical result, however, is not particularly helpful, since the time required to ensure a significant probability of success will usually exceed the time required for a complete search of the solution space.

Pseudocode

The following pseudocode implements the simulated annealing heuristic as described above. It starts from a state `s0` and continues to either a maximum of `kmax` steps or until a state with an energy of `emax` or less is found. In the process, the call `neighbour(s)`

should generate a randomly chosen neighbour of a given state s; the call `random()` should return a random value in the range [0,1]. The annealing schedule is defined by the call `temp(r)`, which should yield the temperature to use, given the fraction `r` of the time budget that has been expended so far.

```
s ← s0; e ← E(s)                                    // Initial state,
energy.
sbest ← s; ebest ← e                                // Initial "best"
solution
k ← 0                                               // Energy evaluation
count.
while k < kmax and e < emax                         // While time left &
not good enough:
  snew ← neighbour(s)                               // Pick some neighbour.
  enew ← E(snew)                                    // Compute its energy.
  if P(e, enew, temp(k/kmax)) > random() then       // Should we move to
it?
    s ← snew; e ← enew                              // Yes, change state.
  if enew < ebest then                              // Is this a new best?
    sbest ← snew; ebest ← enew                      // Save 'new neighbour'
to 'best found'.
  k ← k + 1                                         // One more evaluation
done
return sbest                                        // Return the best
solution found.
```

Actually, the "pure" SA algorithm does not keep track of the best solution found so far: it does not use the variables `sbest` and `ebest`, it lacks the second **`if`** inside the loop, and, at the end, it returns the current state `s` instead of `sbest`. While remembering the best state is a standard technique in optimization that can be used in any metaheuristic, it does not have an analogy with physical annealing — since a physical system can "store" a single state only.

In strict mathematical terms, saving the best state is not necessarily an improvement, since one may have to specify a smaller `kmax` in order to compensate for the higher cost per iteration and since there is a good probability that `sbest` equals `s` in the final iteration anyway. However, the step `sbest ← snew` happens only on a small fraction of the moves. Therefore, the optimization is usually worthwhile, even when state-copying is an expensive operation.

Selecting the parameters

In order to apply the SA method to a specific problem, one must specify the following parameters: the state space, the energy (goal) function `E()`, the candidate generator procedure `neighbour()`, the acceptance probability function `P()`, and the annealing schedule `temp()` AND initial temperature <init temp>. These choices can have a significant impact on the method's effectiveness. Unfortunately, there are no choices of these parameters that will be good for all problems, and there is no general way to find

the best choices for a given problem. The following sections give some general guidelines.

Diameter of the search graph

Simulated annealing may be modeled as a random walk on a *search graph*, whose vertices are all possible states, and whose edges are the candidate moves. An essential requirement for the `neighbour()` function is that it must provide a sufficiently short path on this graph from the initial state to any state which may be the global optimum. (In other words, the diameter of the search graph must be small.) In the traveling salesman example above, for instance, the search space for $n = 20$ cities has $n! = 2432902008176640000$ (2.4 quintillion) states; yet the neighbour generator function that swaps two consecutive cities can get from any state (tour) to any other state in maximum $n(n - 1) / 2 = 190$ steps.

Transition probabilities

For each edge (s,s') of the search graph, one defines a *transition probability*, which is the probability that the SA algorithm will move to state s' when its current state is s. This probability depends on the current temperature as specified by `temp()`, by the order in which the candidate moves are generated by the `neighbour()` function, and by the acceptance probability function `P()`. (Note that the transition probability is **not** simply $P(e,e',T)$, because the candidates are tested serially.)

Acceptance probabilities

The specification of `neighbour()`, `P()`, and `temp()` is partially redundant. In practice, it's common to use the same acceptance function `P()` for many problems, and adjust the other two functions according to the specific problem.

In the formulation of the method by Kirkpatrick et al., the acceptance probability function $P(e,e',T)$ was defined as 1 if $e' < e$, and $\exp((e - e') / T)$ otherwise. This formula was superficially justified by analogy with the transitions of a physical system; it corresponds to the Metropolis-Hastings algorithm, in the case where the proposal distribution of Metropolis-Hastings is symmetric. However, this acceptance probability is often used for simulated annealing even when the `neighbour()` function, which is analogous to the proposal distribution in Metropolis-Hastings, is not symmetric, or not probabilistic at all. As a result, the transition probabilities of the simulated annealing algorithm do not correspond to the transitions of the analogous physical system, and the long-term distribution of states at a constant temperature T need not bear any resemblance to the thermodynamic equilibrium distribution over states of that physical system, at any temperature. Nevertheless, most descriptions of SA assume the original acceptance function, which is probably hard-coded in many implementations of SA.

Efficient candidate generation

When choosing the candidate generator `neighbour()`, one must consider that after a few iterations of the SA algorithm, the current state is expected to have much lower energy than a random state. Therefore, as a general rule, one should skew the generator towards candidate moves where the energy of the destination state *s*' is likely to be similar to that of the current state. This heuristic (which is the main principle of the Metropolis-Hastings algorithm) tends to exclude "very good" candidate moves as well as "very bad" ones; however, the latter are usually much more common than the former, so the heuristic is generally quite effective.

In the traveling salesman problem above, for example, swapping two *consecutive* cities in a low-energy tour is expected to have a modest effect on its energy (length); whereas swapping two *arbitrary* cities is far more likely to increase its length than to decrease it. Thus, the consecutive-swap neighbour generator is expected to perform better than the arbitrary-swap one, even though the latter could provide a somewhat shorter path to the optimum (with $n - 1$ swaps, instead of $n(n - 1) / 2$).

A more precise statement of the heuristic is that one should try first candidate states *s*' for which $P(E(s),E(s'),T)$ is large. For the "standard" acceptance function P above, it means that $E(s') - E(s)$ is on the order of T or less. Thus, in the traveling salesman example above, one could use a `neighbour()` function that swaps two random cities, where the probability of choosing a city pair vanishes as their distance increases beyond T.

Barrier avoidance

When choosing the candidate generator `neighbour()` one must also try to reduce the number of "deep" local minima — states (or sets of connected states) that have much lower energy than all its neighbouring states. Such "closed catchment basins" of the energy function may trap the SA algorithm with high probability (roughly proportional to the number of states in the basin) and for a very long time (roughly exponential on the energy difference between the surrounding states and the bottom of the basin).

As a rule, it is impossible to design a candidate generator that will satisfy this goal and also prioritize candidates with similar energy. On the other hand, one can often vastly improve the efficiency of SA by relatively simple changes to the generator. In the traveling salesman problem, for instance, it is not hard to exhibit two tours A, B, with nearly equal lengths, such that (0) A is optimal, (1) every sequence of city-pair swaps that converts A to B goes through tours that are much longer than both, and (2) A can be transformed into B by flipping (reversing the order of) a set of consecutive cities. In this example, A and B lie in different "deep basins" if the generator performs only random pair-swaps; but they will be in the same basin if the generator performs random segment-flips.

Cooling schedule

The physical analogy that is used to justify SA assumes that the cooling rate is low enough for the probability distribution of the current state to be near thermodynamic equilibrium at all times. Unfortunately, the *relaxation time*—the time one must wait for the equilibrium to be restored after a change in temperature—strongly depends on the "topography" of the energy function and on the current temperature. In the SA algorithm, the relaxation time also depends on the candidate generator, in a very complicated way. Note that all these parameters are usually provided as black box functions to the SA algorithm.

Therefore, in practice the ideal cooling rate cannot be determined beforehand, and should be empirically adjusted for each problem. The variant of SA known as thermodynamic simulated annealing tries to avoid this problem by dispensing with the cooling schedule, and instead automatically adjusting the temperature at each step based on the energy difference between the two states, according to the laws of thermodynamics.

Restarts

Sometimes it is better to move back to a solution that was significantly better rather than always moving from the current state. This process is called *restarting* of simulated annealing. To do this we set `s` and `e` to `sbest` and `ebest` and perhaps restart the annealing schedule. The decision to restart could be based on several criteria. notable among these include restarting based a fixed number of steps, based on whether the current energy being too high from the best energy obtained so far, restarting randomly etc.

Related methods

- Quantum annealing uses "quantum fluctuations" instead of thermal fluctuations to get through high but thin barriers in the target function.
- Stochastic tunneling attempts to overcome the increasing difficulty simulated annealing runs have in escaping from local minima as the temperature decreases, by 'tunneling' through barriers.
- Tabu search normally moves to neighbouring states of lower energy, but will take uphill moves when it finds itself stuck in a local minimum; and avoids cycles by keeping a "taboo list" of solutions already seen.
- Stochastic gradient descent runs many greedy searches from random initial locations.
- Genetic algorithms maintain a pool of solutions rather than just one. New candidate solutions are generated not only by "mutation" (as in SA), but also by

"combination" of two solutions from the pool. Probabilistic criteria, similar to those used in SA, are used to select the candidates for mutation or combination, and for discarding excess solutions from the pool.

- Graduated optimization digressively "smooths" the target function while optimizing.
- Ant colony optimization (ACO) uses many ants (or agents) to traverse the solution space and find locally productive areas.
- The cross-entropy method (CE) generates candidates solutions via a parameterized probability distribution. The parameters are updated via cross-entropy minimization, so as to generate better samples in the next iteration.
- Harmony search mimics musicians in improvisation process where each musician plays a note for finding a best harmony all together.
- Stochastic optimization is an umbrella set of methods that includes simulated annealing and numerous other approaches.
- Particle swarm optimization is an algorithm modelled on swarm intelligence that finds a solution to an optimization problem in a search space, or model and predict social behavior in the presence of objectives.
- Intelligent Water Drops (IWD) which mimics the behavior of natural water drops to solve optimization problems
- Parallel tempering is a simulation of model copies at different temperatures (or Hamiltonians) to overcome the potential barriers.

Chapter- 2

Ant Colony Optimization

Ant behavior was the inspiration for the metaheuristic optimization technique

In computer science and operations research, the **ant colony optimization** algorithm **(ACO)** is a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs.

This algorithm is a member of **ant colony algorithms** family, in swarm intelligence methods, and it constitutes some metaheuristic optimizations. Initially proposed by

Marco Dorigo in 1992 in his PhD thesis, the first algorithm was aiming to search for an optimal path in a graph, based on the behavior of ants seeking a path between their colony and a source of food. The original idea has since diversified to solve a wider class of numerical problems, and as a result, several problems have emerged, drawing on various aspects of the behavior of ants.

Overview

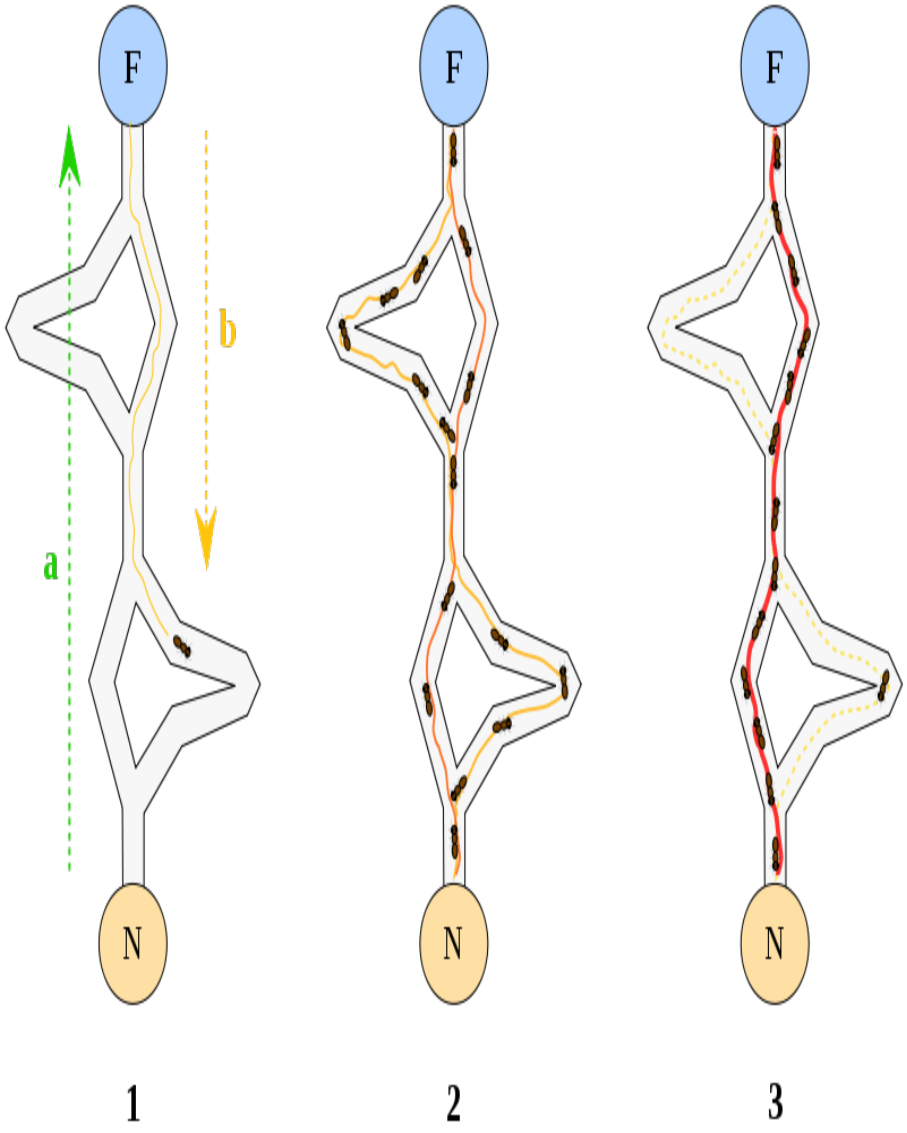
Summary

In the real world, ants (initially) wander randomly, and upon finding food return to their colony while laying down pheromone trails. If other ants find such a path, they are likely not to keep travelling at random, but to instead follow the trail, returning and reinforcing it if they eventually find food.

Over time, however, the pheromone trail starts to evaporate, thus reducing its attractive strength. The more time it takes for an ant to travel down the path and back again, the more time the pheromones have to evaporate. A short path, by comparison, gets marched over faster, and thus the pheromone density remains high as it is laid on the path as fast as it can evaporate. Pheromone evaporation has also the advantage of avoiding the convergence to a locally optimal solution. If there were no evaporation at all, the paths chosen by the first ants would tend to be excessively attractive to the following ones. In that case, the exploration of the solution space would be constrained.

Thus, when one ant finds a good (i.e., short) path from the colony to a food source, other ants are more likely to follow that path, and positive feedback eventually leads all the ants following a single path. The idea of the ant colony algorithm is to mimic this behavior with "simulated ants" walking around the graph representing the problem to solve.

Detailed



The original idea comes from observing the exploitation of food resources among ants, in which ants' individually limited cognitive abilities have collectively been able to find the shortest path between a food source and the nest.

1. The first ant finds the food source (F), via any way (a), then returns to the nest (N), leaving behind a trail pheromone (b)
2. Ants indiscriminately follow four possible ways, but the strengthening of the runway makes it more attractive as the shortest route.
3. Ants take the shortest route, long portions of other ways lose their trail pheromones.

In a series of experiments on a colony of ants with a choice between two unequal length paths leading to a source of food, biologists have observed that ants tended to use the shortest route. A model explaining this behaviour is as follows:

1. An ant (called "blitz") runs more or less at random around the colony;
2. If it discovers a food source, it returns more or less directly to the nest, leaving in its path a trail of pheromone;
3. These pheromones are attractive, nearby ants will be inclined to follow, more or less directly, the track;
4. Returning to the colony, these ants will strengthen the route;
5. If there are two routes to reach the same food source then, in a given amount of time, the shorter one will be traveled by more ants than the long route;
6. The short route will be increasingly enhanced, and therefore become more attractive;
7. The long route will eventually disappear because pheromones are volatile;
8. Eventually, all the ants have determined and therefore "chosen" the shortest route.

Ants use the environment as a medium of communication. They exchange information indirectly by depositing pheromones, all detailing the status of their "work". The information exchanged has a local scope, only an ant located where the pheromones were left has a notion of them. This system is called "Stigmergy" and occurs in many social animal societies (it has been studied in the case of the construction of pillars in the nests of termites). The mechanism to solve a problem too complex to be addressed by single ants is a good example of a self-organized system. This system is based on positive feedback (the deposit of pheromone attracts other ants that will strengthen it themselves) and negative (dissipation of the route by evaporation prevents the system from thrashing). Theoretically, if the quantity of pheromone remained the same over time on all edges, no route would be chosen. However, because of feedback, a slight variation on an edge will be amplified and thus allow the choice of an edge. The algorithm will move from an unstable state in which no edge is stronger than another, to a stable state where the route is composed of the strongest edges.

The basic philosophy of the algorithm involves the movement of a colony of ants through the different states of the problem influenced by two local decision policies, viz., *trails* and *attractiveness*. Thereby, each such ant incrementally constructs a solution to the problem. When an ant completes a solution, or during the construction phase, the ant evaluates the solution and modifies the trail value on the components used in its solution. This pheromone information will direct the search of the future ants. Furthermore, the algorithm also includes two more mechanisms, viz., *trail evaporation* and *daemon actions*. *Trail evaporation* reduces all trail values over time thereby avoiding any possibilities of getting stuck in local optima. The *daemon actions* are used to bias the search process from a non-local perspective.

Common extensions

Here are some of most popular variations of ACO Algorithms

Elitist ant system

The global best solution deposits pheromone on every iteration along with all the other ants

Max-Min ant system (MMAS)

Added Maximum and Minimum pheromone amounts [τ_{max},τ_{min}] Only global best or iteration best tour deposited pheromone All edges are initialized to τ_{max} and reinitialized to τ_{max} when nearing stagnation.

Ant Colony System

It has been presented above.

Rank-based ant system (ASrank)

All solutions are ranked according to their fitness. The amount of pheromone deposited is then weighted for each solution, such that the solutions with better fitness deposit more pheromone than the solutions with worse fitness.

Continuous orthogonal ant colony (COAC)

The pheromone deposit mechanism of COAC is to enable ants to search for solutions collaboratively and effectively. By using an orthogonal design method, ants in the feasible domain can explore their chosen regions rapidly and efficiently, with enhanced global search capability and accuracy.

The orthogonal design method and the adaptive radius adjustment method can also be extended to other optimization algorithms for delivering wider advantages in solving practical problems.

Convergence

For some versions of the algorithm, it is possible to prove that it is convergent (i.e. it is able to find the global optimum in a finite time). The first evidence of a convergence ant colony algorithm was made in 2000, the graph-based ant system algorithm, and then algorithms for ACS and MMAS. Like most metaheuristics, it is very difficult to estimate the theoretical speed of convergence. In 2004, Zlochin and his colleagues have shown COA type algorithms could be assimilated methods of stochastic gradient descent, on the cross-entropy and Estimation of distribution algorithm. They proposed that these metaheuristics as a "research-based model".

Example pseudo-code and formulae

```
procedure ACO_MetaHeuristic
  while(not_termination)
     generateSolutions()
     daemonActions()
     pheromoneUpdate()
  end while
end procedure
```

Edge selection

An ant is a simple computational agent in the ant colony optimization algorithm. It iteratively constructs a solution for the problem at hand. The intermediate solutions are referred to as solution states. At each iteration of the algorithm, each ant moves from a state x to state y, corresponding to a more complete intermediate solution. Thus, each ant k computes a set $A_k(x)$ of feasible expansions to its current state in each iteration, and moves to one of these in probability. For ant k, the probability p_{xy}^k of moving from state x to state y depends on the combination of two values, viz., the *attractiveness* η_{xy} of the move, as computed by some heuristic indicating the *a priori* desirability of that move and the *trail level* τ_{xy} of the move, indicating how proficient it has been in the past to make that particular move.

The *trail level* represents a posteriori indication of the desirability of that move. Trails are updated usually when all ants have completed their solution, increasing or decreasing the level of trails corresponding to moves that were part of "good" or "bad" solutions, respectively.

In general, the kth ant moves from state x to state y with probability

$$p_{xy}^k = \frac{(\tau_{xy}^\alpha)(\eta_{xy}^\beta)}{\sum(\tau_{xy}^\alpha)(\eta_{xy}^\beta)}$$

where

τ_{xy} is the amount of pheromone deposited for transition from state x to y, $0 \leq \alpha$ is a parameter to control the influence of τ_{xy}, η_{xy} is the desirability of state transition xy (*a priori* knowledge, typically $1 / d_{xy}$, where d is the distance) and $\beta \leq 1$ is a parameter to control the influence of η_{xy}.

Pheromone update

When all the ants have completed a solution, the trails are updated by

$$\tau_{xy}^k = (1-\rho)\tau_{xy}^k + \Delta\tau_{xy}^k$$

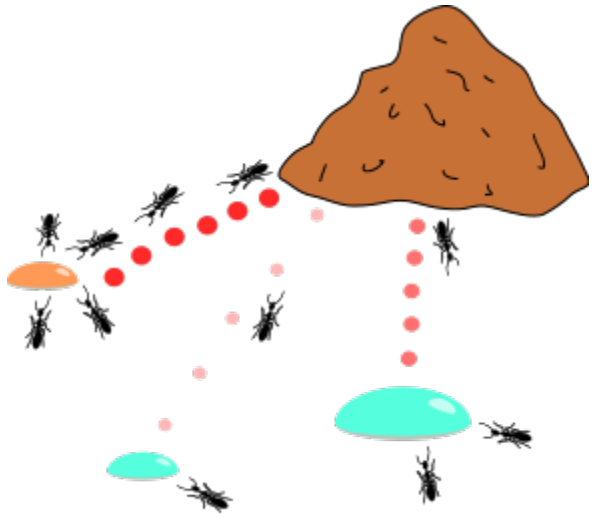
where

τ_{xy}^{k} is the amount of pheromone deposited for a state transition *xy*, ρ is the *pheromone evaporation coefficient* and $\Delta\tau_{i,j}^{k}$ is the amount of pheromone deposited, typically given for a TSP problem (with moves corresponding to arcs of the graph) by

$$\Delta\tau_{xy}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ uses curve } xy \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

where L_k is the cost of the *k*th ant's tour (typically length) and Q is a constant.

Applications



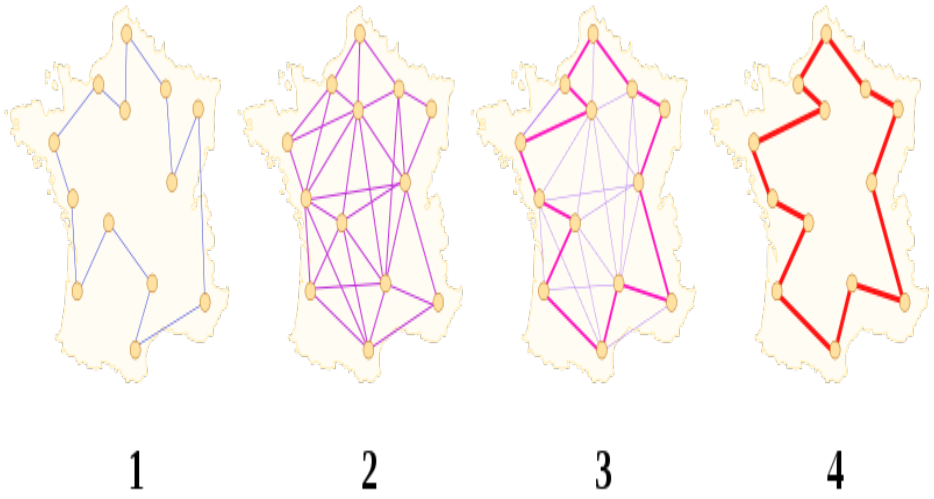
Knapsack problem. The ants prefer the smaller drop of honey over the more abundant, but less nutritious, sugar.

Ant colony optimization algorithms have been applied to many combinatorial optimization problems, ranging from quadratic assignment to fold protein or routing vehicles and a lot of derived methods have been adapted to dynamic problems in real variables, stochastic problems, multi-targets and parallel implementations. It has also been used to produce near-optimal solutions to the travelling salesman problem. They have an advantage over simulated annealing and genetic algorithm approaches of similar problems when the graph may change dynamically; the ant colony algorithm can be run continuously and adapt to changes in real time. This is of interest in network routing and urban transportation systems.

As a very good example, ant colony optimization algorithms have been used to produce near-optimal solutions to the travelling salesman problem. The first ACO algorithm was

called the Ant system and it was aimed to solve the travelling salesman problem, in which the goal is to find the shortest round-trip to link a series of cities. The general algorithm is relatively simple and based on a set of ants, each making one of the possible round-trips along the cities. At each stage, the ant chooses to move from one city to another according to some rules:

1. It must visit each city exactly once;
2. A distant city has less chance of being chosen (the visibility);
3. The more intense the pheromone trail laid out on an edge between two cities, the greater the probability that that edge will be chosen;
4. Having completed its journey, the ant deposits more pheromones on all edges it traversed, if the journey is short;
5. After each iteration, trails of pheromones evaporate.



Scheduling problem

- Job-shop scheduling problem (JSP)
- Open-shop scheduling problem (OSP)
- Permutation flow shop problem (PFSP)
- Single machine total tardiness problem (SMTTP)
- Single machine total weighted tardiness problem (SMTWTP)
- Resource-constrained project scheduling problem (RCPSP)
- Group-shop scheduling problem (GSP)
- Single-machine total tardiness problem with sequence dependent setup times (SMTTPDST)
- Multistage Flowshop Scheduling Problem (MFSP) with sequence dependent setup/changeover times

Vehicle routing problem

- Capacitated vehicle routing problem (CVRP)
- Multi-depot vehicle routing problem (MDVRP)
- Period vehicle routing problem (PVRP)
- Split delivery vehicle routing problem (SDVRP)
- Stochastic vehicle routing problem (SVRP)
- Vehicle routing problem with pick-up and delivery (VRPPD)
- Vehicle routing problem with time windows (VRPTW)
- Time Dependent Vehicle Routing Problem with Time Windows (TDVRPTW)

Assignment problem

- Quadratic assignment problem (QAP)
- Generalized assignment problem (GAP)
- Frequency assignment problem (FAP)
- Redundancy allocation problem (RAP)

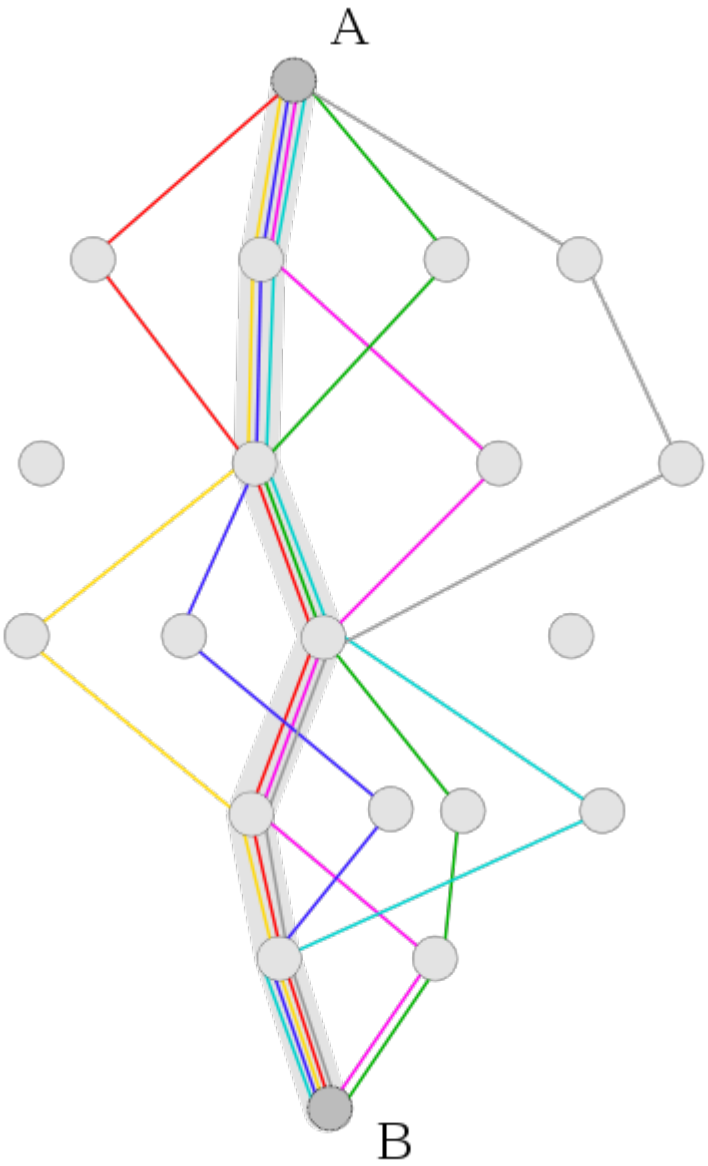
Set problem

- Set covering problem(SCP)
- Set partition problem (SPP)
- Weight constrained graph tree partition problem (WCGTPP)
- Arc-weighted l-cardinality tree problem (AWlCTP)
- Multiple knapsack problem (MKP)
- Maximum independent set problem (MIS)

Others

- Classification
- Connection-oriented network routing
- Connectionless network routing
- Data mining
- Discounted cash flows in project scheduling
- Grid Workflow Scheduling Problem
- Image processing
- Intelligent testing system
- System identification
- Protein Folding
- Power Electronic Circuit Design

Definition difficulty



e

With an ACO algorithm, the shortest path in a graph, between two points A and B, is built from a combination of several paths. It is not easy to give a precise definition of what algorithm is or is not an ant colony, because the definition may vary according to the authors and uses. Broadly speaking, ant colony algorithms are regarded as populated metaheuristics with each solution represented by an ant moving in the search space. Ants mark the best solutions and take account of previous markings to optimize their search. They can be seen as probabilistic multi-agent algorithms using a probability distribution to make the transition between each iteration. In their versions for combinatorial problems, they use an iterative construction of solutions. According to some authors, the

thing which distinguishes ACO algorithms from other relatives (such as algorithms to estimate the distribution or particle swarm optimization) is precisely their constructive aspect. In combinatorial problems, it is possible that the best solution eventually be found, even though no ant would prove effective. Thus, in the example of the Travelling salesman problem, it is not necessary that an ant actually travels the shortest route: the shortest route can be built from the strongest segments of the best solutions. However, this definition can be problematic in the case of problems in real variables, where no structure of 'neighbours' exists. The collective behaviour of social insects remains a source of inspiration for researchers. The wide variety of algorithms (for optimization or not) seeking self-organization in biological systems has led to the concept of "swarm intelligence", which is a very general framework in which ant colony algorithms fit.

Stigmergy algorithms

There is in practice a large number of algorithms claiming to be "ant colonies", without always sharing the general framework of optimization by canonical ant colonies (COA). In practice, the use of an exchange of information between ants via the environment (a principle called "Stigmergy") is deemed enough for an algorithm to belong to the class of ant colony algorithms. This principle has led some authors to create the term "value" to organize methods and behavior based on search of food, sorting larvae, division of labour and cooperative transportation.

Chapter- 3

Stochastic Optimization

Stochastic optimization (SO) methods are optimization algorithms which incorporate probabilistic (random) elements, either in the problem data (the objective function, the constraints, etc.), or in the algorithm itself (through random parameter values, random choices, etc.), or in both. The concept contrasts with the deterministic optimization methods, where the values of the objective function are assumed to be exact, and the computation is completely determined by the values sampled so far.

Methods for stochastic functions

Partly-random input data arise in such areas as real-time estimation and control, simulation-based optimization where Monte Carlo simulations are run as estimates of an actual system, and problems where there is experimental (random) error in the measurements of the criterion. In such cases, knowledge that the function values are contaminated by random "noise" leads naturally to algorithms that use statistical inference tools to estimate the "true" values of the function and/or make statistically optimal decisions about the next steps. Methods of this class include

Stochastic approximation

Stochastic approximation methods are a family of iterative stochastic optimization algorithms that attempt to find zeroes or extrema of functions which cannot be computed directly, but only estimated via noisy observations. The first, and prototypical, algorithms of this kind were the **Robbins-Monro** and **Kiefer-Wolfowitz** algorithms.

Robbins-Monro algorithm

In the Robbins-Monro algorithm, introduced in 1951, one has a function $M(x)$ for which one wishes to find the value of x, x_0, satisfying $M(x_0) = \alpha$. However, what is observable is not $M(x)$, but rather a random variable $N(x)$ such that $E(N(x) \mid x) = M(x)$. The algorithm is then to construct a sequence $x_1, x_2, \cdots$ which satisfies

$$x_{n+1} = x_n + a_n(\alpha - N(x_n)).$$

Here, $a_1, a_2, \cdots$ is a sequence of positive step-sizes. Robbins and Monro proved that, if $N(x)$ is uniformly bounded, $M(x)$ is nondecreasing, $M'(x_0)$ exists and is positive, and if a_n satisfies a set of bounds (fulfilled if one takes $a_n = 1 / n$), then x_n converges in L^2 (and hence also in probability) to x_0.[Theorem 2]. In general, the a_n'*s* need not equal $1 / n$. However, to ensure convergence, they should converge to zero, and in order to average out the noise in $N(x)$, they should converge slowly.

Kiefer-Wolfowitz algorithm

In the Kiefer-Wolfowitz algorithm, introduced a year after the Robbins-Monro algorithm, one wishes to find the maximum, x_0, of the unknown $M(x)$ and constructs a sequence $x_1, x_2, \cdots$ such that

$$x_{n+1} = x_n + a_n \frac{N(x_n + c_n) - N(x_n - c_n)}{c_n}.$$

Here, $a_1, a_2, \cdots$ is a sequence of positive step sizes which serve the same function as in the Robbins-Monro algorithm, and $c_1, c_2, \cdots$ is a sequence of positive step sizes which are used to estimate, via finite differences, the derivative of M. Kiefer and Wolfowitz showed that, if a_n and c_n satisfy various bounds (fulfilled by taking $a_n = 1 / n$, $c_n = (1 / n)^{1/3}$), and $M(x)$ and $N(x)$ satisfy some technical conditions, then the sequence x_n converges in probability to x_0.

Subsequent developments

An extensive theoretical literature has grown up around these algorithms, concerning conditions for convergence, rates of convergence, multivariate and other generalizations, proper choice of step size, possible noise models, and so on. These methods are also applied in control theory, in which case the unknown function which we wish to optimize or find the zero of may vary in time. In this case, the step size a_n should not converge to zero but should be chosen so as to track the function.

Stochastic gradient descent

Stochastic gradient descent is an optimization method for minimizing an objective function that is written as a sum of differentiable functions.

Background

Both statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum:

$$Q(w) = \sum_{i=1}^{n} Q_i(w),$$

where the parameter *w* is to be estimated and where typically each summand function $Q_i()$ is associated with the *i*-th observation in the data set (used for training).

In classical statistics, sum-minimization problems arise in least squares and of maximum-likelihood estimation (for independent observations). The general class of estimators that arise as minimizers of sums are called M-estimators. However, in statistics, it has been long recognized that requiring even local minimization is too restrictive for some problems of maximum-likelihood estimation, as shown for example by Thomas Ferguson's example. Therefore, contemporary statisticial theorists often consider stationary points of the likelihood function (or zeros of its derivative, the score function, and other estimating equations).

The sum-minimization problem also arises for empirical risk minimization: In this case, $Q_i(w)$ is the value of loss function at *i*-th example, and $Q(w)$ is the empirical risk.

When used to minimize the above function, a standard (or "batch") gradient descent method would perform the following iterations :

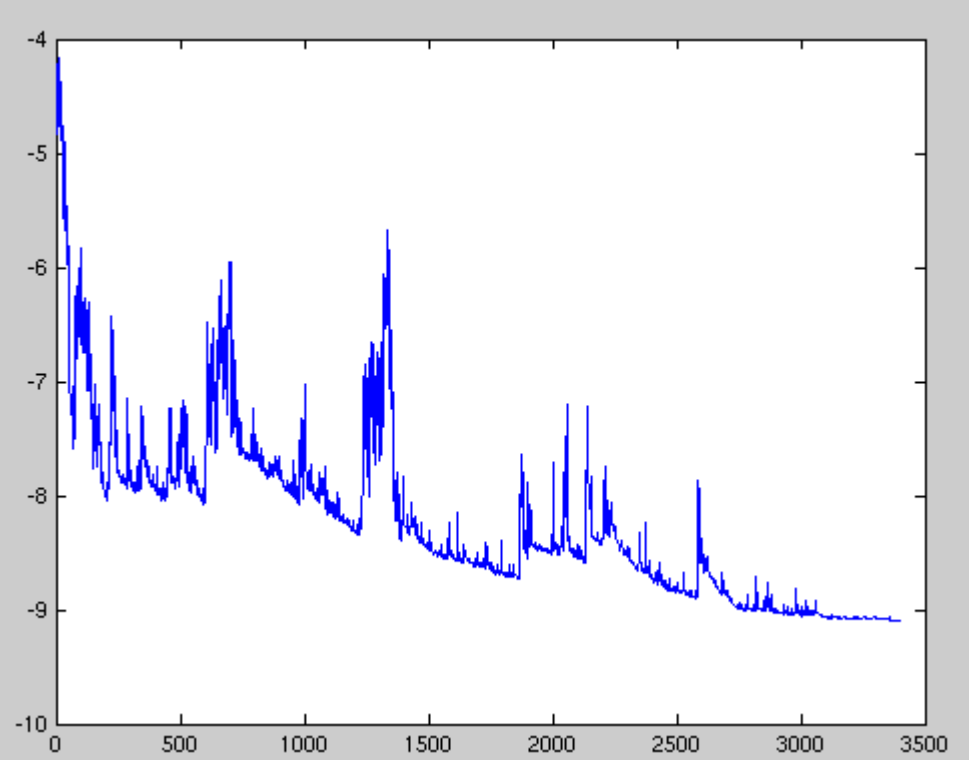
$$w := w - \alpha \nabla Q(w) = w - \alpha \sum_{i=1}^{n} \nabla Q_i(w),$$

where α is a step size (sometimes called the *learning rate* in machine learning).

In many cases, the summand functions have a simple form that enables inexpensive evaluations of the sum-function and the sum gradient. For example, in statistics, one-parameter exponential families allow economical function-evaluations and gradient-evaluations.

However, in other cases, evaluating the sum-gradient may require expensive evaluations of the gradients from all summand functions. When the training set is enormous and no simple formulas exist, evaluating the sums of gradients becomes very expensive, because evaluating the gradient requires evaluating all the summand functions' gradients. To economize on the computational cost at every iteration, stochastic gradient descent samples a subset of summand functions at every step.

Iterative method



Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

In stochastic (or "on-line") gradient descent, the true gradient of $Q(w)$ is approximated by a gradient at a single example:

$$w := w - \alpha \nabla Q_i(w).$$

As the algorithm sweeps through the training set, it performs the above update for each training example. Several passes over the training set are made until the algorithm converges. Typical implementations may also randomly shuffle training examples at each pass and use an adaptive learning rate.

In pseudocode, stochastic gradient descent with shuffling of training set at each pass can be presented as follows:

- Choose an initial vector of parameters w and learning rate α.
- Repeat until an approximate minimum is obtained:
  - Randomly shuffle examples in the training set.
  - For $i = 1, 2, ..., n$, do:
    - $w := w - \alpha \nabla Q_i(w).$

There is a compromise between the two forms, which is often called "mini-batches", where the true gradient is approximated by a sum over a small number of training examples.

The convergence of stochastic gradient descent has been analyzed using the theories of convex minimization and of stochastic approximation. Briefly, stochastic gradient methods need not converge to a global minimum unless the objective function is convex --- or more generally unless the problem has convex lower level sets and the function has the monotone property called "pseudoconvexity" in optimization theory, and finally the step-sizes are very small).

Example

Let's suppose we want to fit a straight line $y = w_1 + w_2 x$ to a training set of two-dimensional points $(x_1, y_1), \ldots, (x_n, y_n)$ using least squares. The objective function to be minimized is:

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{n} \left(w_1 + w_2 x_i - y_i\right)^2.$$

The last line in the above pseudocode for this specific problem will become:

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} := \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} - \alpha 2 (w_1 + w_2 x_i - y_i) \begin{pmatrix} 1 \\ x_i \end{pmatrix}.$$

Applications

Some of the most popular stochastic gradient descent algorithms are the least mean squares (LMS) adaptive filter and the backpropagation algorithm.

Simultaneous perturbation stochastic approximation

Simultaneous perturbation stochastic approximation (SPSA) is an algorithmic method for optimizing systems with multiple unknown parameters. It is a type of stochastic approximation algorithm. As an optimization method, it is appropriately suited to large-scale population models, adaptive modeling, simulation optimization, and atmospheric modeling.

SPSA is a descent method capable of finding global minima. Its main feature is the gradient approximation that requires only two measurements of the objective function, regardless of the dimension of the optimization problem. Recall that we want to find the

optimal control u^* with loss function $J(u)$:

$$h^* = \arg\min\{u \in U\}.$$

Both Finite Differences Stochastic Approximation (FDSA) and SPSA use the same iterative process:

$$u_{n+1} = u_n - a_n \hat{g}_n(u_n).$$

where $u_n = ((u_n)_1,(u_n)_2,,,(u_n)_p)^T$ represents the n^th iterate, $\hat{g}_n(u_n)$ is the estimate of the gradient of the objective function $g(u) = \frac{\partial}{\partial u} J(u)$ evaluated at u_n, and $\{a_n\}$ is a positive number sequence converging to 0. If u_n is a p-dimension vector, the i^th component of the symmetric finite difference gradient estimator is:

FD:

$$(\hat{g}_n(u_n))_i = \frac{J(u_n + c_n e_i) - J(u_n - c_n e_i)}{2c_n},$$

$1 \leq i \leq p$ where e_i is the unit vector with a 1 in the i^th place, and cnis a small positive number that decreases with n. With this method, $2p$ evaluations of J for each g_n are needed. Clearly, when p is large, this estimator looses efficiency.

SP:

$$(\hat{g}_n(u_n))_i = \frac{J(u_n + c_n \Delta_n) - J(u_n - c_n \Delta_n)}{2c_n(\Delta_n)_i},$$

Remark that FD perturbs only one direction at the time, while the SP estimator disturbs all directions in the same time (the numerator is identical in all p components). The number of loss function measurements needed in the SPSA method for each g_n is always 2, independent of the dimension p. Thus, SPSA uses p times fewer function evaluations than FDSA, which makes it a lot more efficient.

Simple experiments with $p=2$ showed that SPSA converges in the same number of iterations as FDSA. The latter follows approximately the steepest descent direction, behaving like the gradient method. On the other hand, SPSA, with the random search direction, does not follow exactly the gradient path. In average though, it tracks it nearly because the gradient approximation is an almost unbiased estimator of the gradient, as shown in the following lemma

Lemma 1 *Denote by*

$$b_n = E[\hat{g}_n|\theta_n] - \nabla J(\theta_n)$$

the bias in the estimator ^gn. Assume that $\{(\Delta_n)_i\}$ *are* all mutually independent with zero-mean, bounded second moments, and $E|(\Delta_n)_i^1|$ uniformly bounded on U. Then $b_n \to ()$ w.p. 1.

The main idea is to use conditioning on Δ_n to express $E[(\hat{g}_n)_i]$ and then to use a second order Taylor expansion of $J(u_n + c_n\Delta_n)_i$ | and $J(u_n - c_n\Delta_n)_i$ | . After algebraic manipulations implying the zero mean and the independence of $\{(\Delta_n)_i\}$, we get

$$E[(\hat{g}_n)_i] = (g_n)_i + O(c_n)$$

The result follows from the hypothesis that $c_n \to 0$..

Next we resume some of the hypotheses under which u_t converges in probability to the set of global minima of $J(u)$. The efficiency of the method depends on the shape of $J(u)$,

the values of the parameters a_k and c_k and the distribution of the perturbation terms Δ_{ki}. First, the algorithm parameters must satisfy the following conditions:

- $a_t > 0$, $a_t \to 0$ when $t \to \infty$ and

$\sum_{t=1}^{\infty} a_t = \infty$; good choice would be $a_k = \frac{a}{k}$; a>0;

- $c_t = \frac{c}{t^{\gamma}}$, where c>0, $\gamma \in [\frac{1}{6}, \frac{1}{2}]$;
- $\sum_{t=1}^{\infty} (\frac{a_t}{c_t})^2 < \infty$
- Δ_{ti} must be mutually independent zero-mean random variables, symmetrically distributed about zero, with $\Delta_{ki} < a_1 < \infty$ and $E|\Delta_{ki}^2 + < a_2 < \infty$ a.s., $\forall$ i, k.

A good choice for Δ_{ki} is Bernoulli +-1 with probability 0.5. The uniform and normal distributions do not satisfy the finite moment conditions, so can not be used.
The loss function *J(u)* must be thrice continuously differentiable and the individual elements of the third derivative must be bounded: $|J^{(3)}(u)| < a_3 < \infty$. Also, $|J(u)| \to \infty$ as $u \to \infty$. In addition, ∇J must be Lipschitz continuous, bounded and the ODE $\hat{u} = g(u)$ must have a unique solution for each initial condition. Under these conditions and a few others, u_k converges in probability to the set of global minima of J(u).

Chapter- 4

Stochastic Programming & CMA-ES

Stochastic programming

Stochastic programming is a framework for modeling optimization problems that involve uncertainty. Whereas deterministic optimization problems are formulated with known parameters, real world problems almost invariably include some unknown parameters. When the parameters are known only within certain bounds, one approach to tackling such problems is called robust optimization. Here the goal is to find a solution which is feasible for all such data and optimal in some sense. Stochastic programming models are similar in style but take advantage of the fact that probability distributions governing the data are known or can be estimated. The goal here is to find some policy that is feasible for all (or almost all) the possible data instances and maximizes the expectation of some function of the decisions and the random variables. More generally, such models are formulated, solved analytically or numerically, and analyzed in order to provide useful information to a decision-maker.

As an example, consider two-stage linear programs. Here the decision maker takes some action in the first stage, after which a random event occurs affecting the outcome of the first-stage decision. A recourse decision can then be made in the second stage that compensates for any bad effects that might have been experienced as a result of the first-stage decision. The optimal policy from such a model is a single first-stage policy and a collection of recourse decisions (a decision rule) defining which second-stage action should be taken in response to each random outcome.

Stochastic programming has applications in a broad range of areas ranging from finance to transportation to energy optimization.

Biological Applications

Stochastic dynamic programming is frequently used to model animal behaviour in such fields as behavioural ecology. Empirical tests of models of optimal foraging, life-history transitions such as fledging in birds and egg laying in parasitoid wasps have shown the

value of this modelling technique in explaining the evolution of behavioural decision making. These models are typically many staged, rather than two-staged.

Economic Applications

Stochastic dynamic programming is a useful tool in understanding decision making under uncertainty. The accumulation of capital stock under uncertainty is one example, often it is used by resource economists to analyze bioeconomic problems where the uncertainty enters in such as weather, etc.

Solvers

FortSP is a software package for solving stochastic programming (SP) problems. It solves scenario-based SP problems with recourse as well as problems with chance constraints and integrated chance constraints. FortSP is available as a standalone executable that accepts input in SMPS format and as a library with an interface in the C programming language.

The solution algorithms provided by FortSP include Benders' decomposition and a variant of level decomposition for two-stage problems, nested Benders' decomposition for multistage problems and reformulation of the problem as a deterministic equivalent. There is also an implementation of a cutting-plane algorithm for integrated chance constraints.

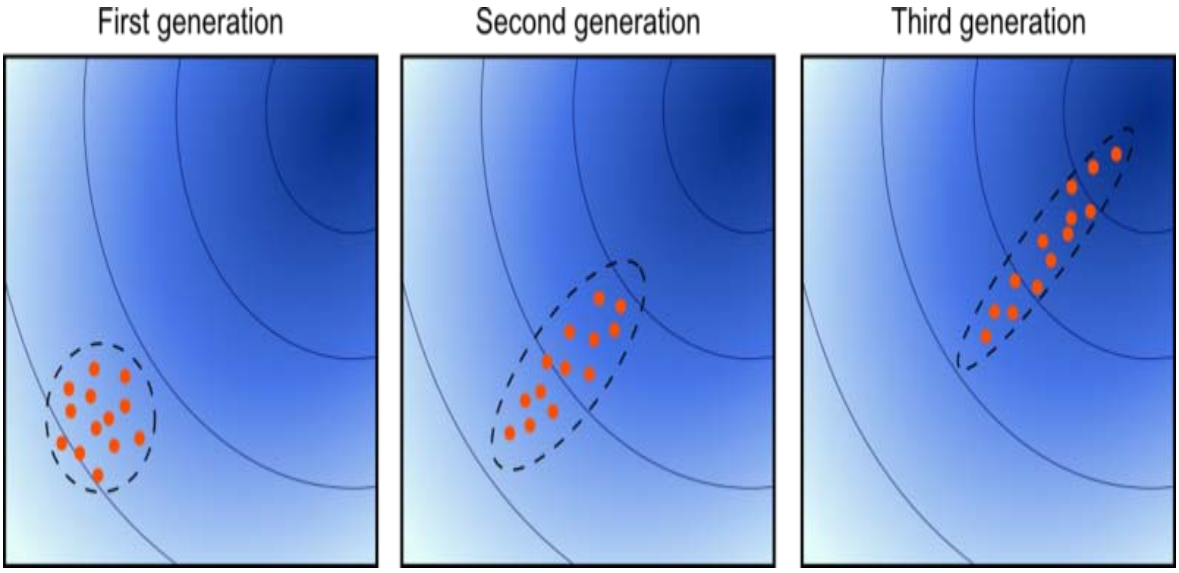
FortSP supports external linear programming solvers such as CPLEX, FortMP and Gurobi through their library interfaces or nl files. These solvers are used to optimize the deterministic equivalent problem and also the subproblems in the decomposition methods.

CMA-ES

CMA-ES stands for Covariance Matrix Adaptation Evolution Strategy. An evolution strategy (ES) is a stochastic, derivative-free numerical optimization method for non-linear or non-convex problems, belonging to the class of evolutionary algorithms and evolutionary computation. In an evolution strategy, new candidate solutions are sampled according to a multivariate normal distribution. Pairwise dependencies between the variables in this distribution are described by a covariance matrix. The covariance matrix adaptation (CMA) is a method to update the covariance matrix of this distribution.

Adaptation of the covariance matrix amounts to learning a second order model of the underlying objective function similar to the approximation of the inverse Hessian matrix in the Quasi-Newton method in classical optimization. In contrast to most classical methods, fewer assumptions on the nature of the underlying objective function are made. Only the ranking between candidate solutions is exploited for learning the sample distribution and neither derivatives nor even the function values themselves are required by the method.

Principles



Concept behind the covariance matrix adaptation. As the generations develop, the distribution shape adapts to an ellipsoidal or ridge-like landscape (the shown landscape is spherical by mistake).

Two main principles for the adaptation of parameters of the search distribution are exploited in the CMA-ES algorithm.

First, a maximum-likelihood principle, based on the idea to increase the probability of successful candidate solutions and search steps. The mean of the distribution is updated such that the likelihood of previously successful candidate solutions is maximized. The covariance matrix of the distribution is updated (incrementally) such that the likelihood of previously successful search steps is increased. Both updates can be interpreted as a natural gradient descent. Also, in consequence, the CMA conducts an iterated principal components analysis of successful search steps while retaining *all* principle axes. Estimation of distribution algorithms and the Cross-Entropy Method are based on very similar ideas, but estimate (non-incrementally) the covariance matrix by maximizing the likelihood of successful solution *points* instead of successful search *steps*.

Second, two paths of the time evolution of the distribution mean of the strategy are recorded, called search or evolution paths. These paths contain significant information

about the correlation between consecutive steps. Specifically, if consecutive steps are taken in a similar direction, the evolution paths become long. The evolution paths are exploited in two ways. One path is used for the covariance matrix adaptation procedure in place of single successful search steps and facilitates a possibly much faster variance increase of favorable directions. The other path is used to conduct an additional step-size control. This step-size control aims to make consecutive movements of the distribution mean orthogonal in expectation. The step-size control effectively prevents premature convergence yet allowing fast convergence to an optimum.

Algorithm

In the following the most commonly used (μ/μ_w, λ)-CMA-ES is outlined, where in each iteration step a weighted combination of the μ best out of λ new candidate solutions is used to update the distribution parameters. For providing an overview, a pseudocode of the algorithm is given.

```
set λ  // number of samples per iteration, at least two, generally > 4
initialize m, σ, C = I, p_σ = 0, p_c = 0  // initialize state variables
while not terminate  // iterate
   for i in {1...λ}  // sample λ new solutions and evaluate them
      x_i = sample_multivariate_normal(mean=m, covariance_matrix=σ^2 C)
      f_i = fitness(x_i)
   x_1...λ ← x_s(1)...s(λ) with s(i) = argsort(f_1...λ, i)  // sort solutions
   m' = m  // we need later m - m' and x_i - m'
   m ← update_m(x_1,..., x_λ)  // move mean to better solutions
   p_σ ← update_ps(p_σ, σ^-1 C^-1/2 (m - m'))  // update isotropic
evolution path
   p_c ← update_pc(p_c, σ^-1 (m - m'),  |  | p_σ |  | )  // update
anisotropic evolution path
   C ← update_C(C, p_c, (x_1 - m') / σ,..., (x_λ - m') / σ)  // update
covariance matrix
   σ ← update_sigma(σ,  |  | p_σ |  | )  // update step-size using
isotropic path length
return m or x_1
```

The main loop consists of three main parts: 1) sampling of new solutions, 2) re-ordering of the sampled solutions based on their fitness, 3) update of the internal state variables based on the re-ordered samples. The order of the updates is important and cannot be changed. State variables and updates are specified in the following, where additionally the iteration index k is introduced.

Given are the search space dimension n and, at iteration step k, the five state variables

$m_k \in \mathbb{R}^n$, the distribution mean and current favorite solution to the optimization problem,
$\sigma_k > 0$, the step-size,
C_k, a symmetric and positive definite $n \times n$covariance matrix with $C_0 = I$ and
$p_\sigma \in \mathbb{R}^n, p_c \in \mathbb{R}^n$, two evolution paths, initially set to the zero vector.

The iteration starts with sampling $\lambda > 1$ candidate solutions $x_i \in \mathbb{R}^n$ from a multivariate normal distribution $\mathcal{N}(m_k, \sigma_k^2 C_k)$, i.e. for $i = 1,...,\lambda$

$$\begin{aligned} x_i &\sim \mathcal{N}(m_k, \sigma_k^2 C_k) \\ &\sim m_k + \sigma_k \times \mathcal{N}(0, C_k) \end{aligned}$$

The second line suggests the interpretation as perturbation (mutation) of the current favorite solution vector m_k (the distribution mean vector). The candidate solutions x_i are evaluated on the objective function $f : \mathbb{R}^n \to \mathbb{R}$ to be minimized. Denoting the *f*-sorted candidate solutions as

$$\{x_{i:\lambda} \mid i = 1 \dots \lambda\} = \{x_i \mid i = 1 \dots \lambda\} \text{ and } f(x_{1:\lambda}) \le \cdots \le f(x_{\mu:\lambda}) \le f(x_{\mu+1:\lambda}) \dots,$$

the new mean value is computed as

$$m_{k+1} = \sum_{i=1}^{\mu} w_i \, x_{i:\lambda}$$

where the positive (recombination) weights $w_1 \ge w_2 \ge \dots \ge w_\mu > 0$ sum to one. Typically, $\mu \le \lambda/2$ and the weights are chosen such that $\mu_w := 1/\sum_{i=1}^{\mu} w_i^2 \approx \lambda/4$. The only feedback used from the objective function here and in the following is an ordering of the sampled candidate solutions due to the indices $i{:}\lambda$.

The step-size σ_k is updated using *cumulative step-size adaptation* (CSA), sometimes also denoted as *path length control*. The evolution path (or search path) p_σ is updated first.

$$p_\sigma \leftarrow \underbrace{(1 - c_\sigma)}_{\text{discount factor}} p_\sigma + \overbrace{\sqrt{1 - (1 - c_\sigma)^2}}^{\text{complements for discounted variance}} \underbrace{\sqrt{\mu_w}\, C_k^{-1/2} \overbrace{\frac{m_{k+1} - m_k}{\sigma_k}}^{\text{displacement of } m}}_{\text{distributed as } \mathcal{N}(0,I) \text{ under neutral selection}}$$

$$\sigma_{k+1} = \sigma_k \times \exp\left(\frac{c_c}{d_c} \underbrace{\left(\frac{\|p_\sigma\|}{E\|\mathcal{N}(0,I)\|} - 1\right)}_{\text{unbiased about 0 under neutral selection}}\right)$$

where

$c_\sigma^{-1} \approx n/3$ is the backward time horizon for the evolution path p_σ and larger than one,

$\mu_w = \left(\sum_{i=1}^{\mu} w_i^2\right)^{-1}$ is the variance effective selection mass and $1 \leq \mu_w \leq \mu$ by definition of w_i,

$C_k^{-1/2} = \sqrt{C_k}^{-1} = \sqrt{C_k^{-1}}$ is the unique symmetric square root of the inverse of C_k, and

d_c is the damping parameter usually close to one. For $d_c = \infty$ or $c_\sigma = 0$ step-size adaptation is off.

The step-size σ_k is increased if and only if $\|p_\sigma\|$ is larger than the expected value

$$E\|\mathcal{N}(0, I)\| = \sqrt{2}\,\Gamma((n+1)/2)/\Gamma(n/2)$$
$$\approx \sqrt{n}\,(1 - 1/(4\,n) + 1/(21\,n^2))$$

and decreased if it is smaller. For this reason, the step-size update tends to make consecutive steps C_k^{-1}-conjugate, in that after the adaptation has been successful $\left(\frac{m_{k+2}-m_{k+1}}{\sigma_{k+1}}\right)^T C_k^{-1} \frac{m_{k+1}-m_k}{\sigma_k} \approx 0$.

Finally, the covariance matrix is updated, where again the respective evolution path is updated first.

$$p_c \leftarrow \underbrace{(1-c_c)}_{\text{discount factor}}\, p_c + \underbrace{1_{[0,\alpha\sqrt{n}]}(\|p_\sigma\|)}_{\text{indicator function}} \overbrace{\sqrt{1-(1-c_c)^2}}^{\text{complements for discounted variance}} \underbrace{\sqrt{\mu_w}\,\frac{m_{k+1}-m_k}{\sigma_k}}_{\text{distributed as } \mathcal{N}(0,C_k) \text{ under neutral selection}}$$

$$C_{k+1} = \underbrace{(1-c_1-c_\mu+c_s)}_{\text{discount factor}}\, C_k + c_1 \underbrace{p_c p_c^T}_{\text{rank one matrix}} + c_\mu \underbrace{\sum_{i=1}^{\mu} w_i \frac{x_{i:\lambda}-m_k}{\sigma_k}\left(\frac{x_{i:\lambda}-m_k}{\sigma_k}\right)^T}_{\text{rank } \min(\mu,n) \text{ matrix}}$$

where T denotes the transpose and

$c_c^{-1} \approx n/4$ is the backward time horizon for the evolution path p_c and larger than one,

$\alpha \approx 1.5$ and the indicator function $\mathbf{1}_{[0,\alpha\sqrt{n}]}(\|p_\sigma\|)$ evaluates to one iff $\|p_\sigma\| \in [0, \alpha\sqrt{n}]$ or, in other words, $\|p_\sigma\| \leq 1.5\sqrt{n}$, which is usually the case,

$c_s = (1 - \mathbf{1}_{[0,\alpha\sqrt{n}]}(\|p_\sigma\|))\, c_1 c_c (2 - c_c)$ makes partly up for the small variance loss in case the indicator is zero,

$c_1 \approx 2/n^2$ is the learning rate for the rank-one update of the covariance matrix and
$c_\mu \approx \mu_w/n^2$ is the learning rate for the rank-μ update of the covariance matrix and must not exceed $1 - c_1$.

The covariance matrix update tends to increase the likelihood for p_c and for $(x_{i:\lambda} - m_k) / \sigma_k$ to be sampled from $\mathcal{N}(0, C_{k+1})$. This completes the iteration step.

The number of candidate samples per iteration, λ, is not determined a priori and can vary in a wide range. Smaller values, for example λ = 10, lead to more local search behavior. Larger values, for example λ = 10*n* with $\mu_w \approx \lambda/4$, render the search more global. Sometimes the algorithm is repeatedly restarted with increasing λ by a factor of two for each restart. Besides of setting λ (or possibly μ instead, if for example λ is predetermined by the number of available processors), the above introduced parameters are not specific to the given objective function and therefore not meant to be modified by the user.

Example code in Matlab/Octave

```
function xmin=purecmaes   % (mu/mu_w, lambda)-CMA-ES

  % --------------------  Initialization --------------------------------
  % User defined input parameters (need to be edited)
  strfitnessfct = 'frosenbrock';  % name of objective/fitness function
  N = 20;               % number of objective variables/problem
dimension
  xmean = rand(N,1);    % objective variables initial point
  sigma = 0.5;          % coordinate wise standard deviation (step
size)
  stopfitness = 1e-10;  % stop if fitness < stopfitness (minimization)
  stopeval = 1e3*N^2;   % stop after stopeval number of function
evaluations

  % Strategy parameter setting: Selection
  lambda = 4+floor(3*log(N));  % population size, offspring number
  mu = lambda/2;               % number of parents/points for
recombination
  weights = log(mu+1/2)-log(1:mu)'; % muXone array for weighted
recombination
  mu = floor(mu);
  weights = weights/sum(weights);     % normalize recombination weights
array
  mueff=sum(weights)^2/sum(weights.^2); % variance-effectiveness of sum
w_i x_i

  % Strategy parameter setting: Adaptation
  cc = (4+mueff/N) / (N+4 + 2*mueff/N);  % time constant for cumulation
for C
  cs = (mueff+2) / (N+mueff+5);  % t-const for cumulation for sigma
control
```

```
  c1 = 2 / ((N+1.3)^2+mueff);    % learning rate for rank-one update of
C
  cmu = 2 * (mueff-2+1/mueff) / ((N+2)^2+mueff);  % and for rank-mu
update
  damps = 1 + 2*max(0, sqrt((mueff-1)/(N+1))-1) + cs; % damping for
sigma
                                                  % usually close
to 1
  % Initialize dynamic (internal) strategy parameters and constants
  pc = zeros(N,1); ps = zeros(N,1);   % evolution paths for C and sigma
  B = eye(N,N);                       % B defines the coordinate system
  D = ones(N,1);                      % diagonal D defines the scaling
  C = B * diag(D.^2) * B';            % covariance matrix C
  invsqrtC = B * diag(D.^-1) * B';    % C^-1/2
  eigeneval = 0;                      % track update of B and D
  chiN=N^0.5*(1-1/(4*N)+1/(21*N^2));  % expectation of
                                      %   ||N(0,I)|| ==
norm(randn(N,1))

  % ------------------- Generation Loop ------------------------------
--
  counteval = 0;  % the next 40 lines contain the 20 lines of
interesting code
  while counteval < stopeval

      % Generate and evaluate lambda offspring
      for k=1:lambda,
          arx(:,k) = xmean + sigma * B * (D .* randn(N,1)); % m + sig *
Normal(0,C)
          arfitness(k) = feval(strfitnessfct, arx(:,k)); % objective
function call
          counteval = counteval+1;
      end

      % Sort by fitness and compute weighted mean into xmean
      [arfitness, arindex] = sort(arfitness); % minimization
      xold = xmean;
      xmean = arx(:,arindex(1:mu))*weights;   % recombination, new mean
value

      % Cumulation: Update evolution paths
      ps = (1-cs)*ps ...
            + sqrt(cs*(2-cs)*mueff) * invsqrtC * (xmean-xold) / sigma;
      hsig = norm(ps)/sqrt(1-(1-cs)^(2*counteval/lambda))/chiN < 1.4 +
2/(N+1);
      pc = (1-cc)*pc ...
            + hsig * sqrt(cc*(2-cc)*mueff) * (xmean-xold) / sigma;

      % Adapt covariance matrix C
      artmp = (1/sigma) * (arx(:,arindex(1:mu))-repmat(xold,1,mu));
      C = (1-c1-cmu) * C ...                  % regard old matrix
           + c1 * (pc*pc' ...                 % plus rank one update
                   + (1-hsig) * cc*(2-cc) * C) ... % minor correction
if hsig==0
           + cmu * artmp * diag(weights) * artmp'; % plus rank mu
update
```

```
      % Adapt step size sigma
      sigma = sigma * exp((cs/damps)*(norm(ps)/chiN - 1));

      % Decomposition of C into B*diag(D.^2)*B' (diagonalization)
      if counteval - eigeneval > lambda/(c1+cmu)/N/10  % to achieve
O(N^2)
          eigeneval = counteval;
          C = triu(C) + triu(C,1)'; % enforce symmetry
          [B,D] = eig(C);           % eigen decomposition,
B==normalized eigenvectors
          D = sqrt(diag(D));        % D is a vector of standard
deviations now
          invsqrtC = B * diag(D.^-1) * B';
      end

      % Break, if fitness is good enough or condition exceeds 1e14,
better termination methods are advisable
      if arfitness(1) <= stopfitness || max(D) > 1e7 * min(D)
          break;
      end

  end % while, end generation loop

  xmin = arx(:, arindex(1)); % Return best point of last iteration.
                             % Notice that xmean is expected to be even
                             % better.

% --------------------------------------------------------------------
function f=frosenbrock(x)
    if size(x,1) < 2 error('dimension must be greater one'); end
    f = 100*sum((x(1:end-1).^2 - x(2:end)).^2) + sum((x(1:end-1)-
1).^2);
```

Theoretical Foundations

Given the distribution parameters—mean, variances and covariances—the normal probability distribution for sampling new candidate solutions is the maximum entropy probability distribution over $\mathbb{R}^n$, that is, the sample distribution with the minimal amount of prior information built into the distribution. More considerations on the update equations of CMA-ES are made in the following.

Variable Metric

The CMA-ES implements a stochastic variable-metric method. In the very particular case of a convex-quadratic objective function

$$f(x) = \tfrac{1}{2}(x - x^*)^T H(x - x^*)$$

the covariance matrix C_k adapts to the inverse of the Hessian matrix H, up to a scalar factor and small random fluctuations. More general, also on the function $g \circ f$, where g is strictly increasing and therefore order preserving and f is from above, the covariance matrix C_k adapts to H^{-1}, up to a scalar factor and small random fluctuations.

Maximum-Likelihood Updates

The update equations for mean and covariance matrix maximize a likelihood while resembling an expectation-maximization algorithm. The update of the mean vector m maximizes a log-likelihood, such that

$$m_{k+1} = \arg\max_m \sum_{i=1}^{\mu} w_i \log p_{\mathcal{N}}(x_{i:\lambda}|m)$$

where

$$\log p_{\mathcal{N}}(x) = -\frac{1}{2}\log\det(2\pi C) - \frac{1}{2}(x-m)^T C^{-1}(x-m)$$

denotes the log-likelihood of x from a multivariate normal distribution with mean m and any positive definite covariance matrix C. To see that m_{k+1} is independent of C remark first that this is the case for any diagonal matrix C, because the coordinate-wise maximizer is independent of a scaling factor. Then, rotation of the data points or choosing C non-diagonal are equivalent.

The rank-μ update of the covariance matrix, that is, the right most summand in the update equation of C_k, maximizes a log-likelihood in that

$$\sum_{i=1}^{\mu} w_i \frac{x_{i:\lambda} - m_k}{\sigma_k} \left(\frac{x_{i:\lambda} - m_k}{\sigma_k}\right)^T = \arg\max_C \sum_{i=1}^{\mu} w_i \log p_{\mathcal{N}}\left(\frac{x_{i:\lambda} - m_k}{\sigma_k}\middle| C\right)$$

for $\mu \geq n$(otherwise C is singular, but substantially the same result holds for $\mu < n$). Here, $p_{\mathcal{N}}(x|C)$denotes the likelihood of x from a multivariate normal distribution with zero mean and covariance matrix C. Therefore, for $c_1 = 0$ and $c_\mu = 1$, C_{k+1} is the above maximum-likelihood estimator.

Natural Gradient Descent in the Space of Sample Distributions

Akimoto et al recently found that the update of the distribution parameters descends in direction of a sampled natural gradient of the expected objective function value $\mathrm{E}\, f(x)$ (to be minimized), where the expectation is taken under the sample distribution. The natural gradient is independent of the parameterization of the distribution. Taken with respect to the parameters θ of the sample distribution p, the gradient of $\mathrm{E}\, f(x)$ can be expressed as

$$\begin{aligned}\nabla_\theta E(f(x)|\theta) &= \nabla_\theta \int_{\mathbb{R}^n} f(x)p(x)\mathrm{d}x \\ &= \int_{\mathbb{R}^n} f(x)p(x)\frac{1}{p(x)}\nabla_\theta p(x)\mathrm{d}x \\ &= \int_{\mathbb{R}^n} f(x)p(x)\nabla_\theta \ln p(x)\mathrm{d}x \\ &= E(f(x)\nabla_\theta \ln p(x|\theta))\end{aligned}$$

where $p(x) = p(x \mid \theta)$ depends on the parameter vector θ, the so-called score function, $\nabla_\theta \ln p(x|\theta)$, indicates the relative sensitivity of p w.r.t. θ, and the expectation is taken with respect to the distribution p. The *natural* gradient of $\mathrm{E}\, f(x)$, complying with the Fisher information metric (an informational distance measure between probability distributions and the curvature of the relative entropy), now reads

$$\tilde{\nabla} E(f(x)|\theta) = F_\theta^{-1} \nabla_\theta E(f(x)|\theta)$$

where the Fisher information matrix F_θ is the expectation of the Hessian of $-\ln p$ and renders the expression independent of the chosen parameterization. Combining the previous equalities we get

$$\begin{aligned}\tilde{\nabla} E(f(x)|\theta) &= F_\theta^{-1} E(f(x)\nabla_\theta \ln p(x|\theta)) \\ &= E(f(x)F_\theta^{-1}\nabla_\theta \ln p(x|\theta))\end{aligned}$$

A Monte Carlo approximation of the latter expectation takes the average over λ samples from p

$$\tilde{\nabla}\widehat{E}_\theta(f) := -\sum_{i=1}^{\lambda} w_i \, F_\theta^{-1}\nabla_\theta \ln p(x_{i:\lambda}|\theta) \quad \text{with } w_i = -f(x_{i:\lambda})/\lambda$$

where the notation $i{:}\lambda$ from above is used and therefore w_i are monotonously decreasing in i. We might use, for a more robust approximation, rather w_i as defined in the CMA-ES and zero for $i > \mu$ and let

$$\theta = [m_k^T \mathrm{vec}(\sigma_k^2 C_k)^T]^T \in \mathbb{R}^{n+n^2}$$

such that $p(. \mid \theta)$ is the density of the multivariate normal distribution $\mathcal{N}(m_k, \sigma_k^2 C_k)$. Then, we have an explicit expression for

$$F_\theta^{-1} = \begin{bmatrix} \sigma_k^2 C_k & 0 \\ 0 & 2\sigma_k^2 C_k \otimes \sigma_k^2 C_k \end{bmatrix}$$

and for

$$\ln p(x|\theta) = \ln p(x|m_k, \sigma_k^2 C_k) = -\frac{1}{2}(x-m_k)^T \sigma_k^{-2} C_k^{-1}(x-m_k) - \frac{1}{2}\ln\det(2\pi\sigma_k^2 C_k)$$

and, after some calculations, the updates in the CMA-ES turn out as

$$\begin{aligned} m_{k+1} &= m_k - \underbrace{[\tilde{\nabla}\widehat{E}_\theta(f)]_{1,\ldots,n}}_{\text{natural gradient for mean}} \\ &= m_k + \sum_{i=1}^{\lambda} w_i (x_{i:\lambda} - m_k) \end{aligned}$$

and

$$\begin{aligned} C_{k+1} &= C_k + c_1(p_c p_c^T - C_k) - \frac{c_\mu}{\sigma_k^2}\,\mathrm{mat}(\overbrace{[\tilde{\nabla}\widehat{E}_\theta(f)]_{n+1,\ldots,n+n^2}}^{\text{natural gradient for covariance matrix}}) \\ &= C_k + c_1(p_c p_c^T - C_k) + \frac{c_\mu}{\sigma_k^2}\sum_{i=1}^{\lambda} w_i \left((x_{i:\lambda} - m_k)(x_{i:\lambda} - m_k)^T - \sigma_k^2 C_k \right) \end{aligned}$$

where mat forms the proper matrix from the respective natural gradient sub-vector. That means, setting $c_1 = c_\sigma = 0$, the CMA-ES updates descend in direction of the approximation $\tilde{\nabla}\widehat{E}_\theta(f)$ of the natural gradient while using different step-sizes (learning rates) for the orthogonal parameters *m* and *C* respectively.

Stationarity or Unbiasedness

It is comparatively easy to see that the update equations of CMA-ES satisfy some stationarity conditions, in that they are essentially unbiased. Under neutral selection, where $x_{i:\lambda} \sim \mathcal{N}(m_k, \sigma_k^2 C_k)$, we find that

$$E(m_{k+1} \mid m_k) = m_k$$

and under some mild additional assumptions on the initial conditions

$$E(\log \sigma_{k+1} \mid \sigma_k) = \log \sigma_k$$

and with an additional minor correction in the covariance matrix update for the case where the indicator function evaluates to zero, we find

$$E(C_{k+1} \mid C_k) = C_k$$

Invariance

Invariance properties imply uniform performance on a class of objective functions. They have been argued to be an advantage, because they allow to generalize and predict the behavior of the algorithm and therefore strengthen the meaning of empirical results obtained on single functions. The following invariance properties have been established for CMA-ES.

- Invariance under order-preserving transformations of the objective function value *f*, in that for any $h : \mathbb{R}^n \to \mathbb{R}$ the behavior is identical on $f : x \mapsto g(h(x))$ for all strictly increasing $g : \mathbb{R} \to \mathbb{R}$. This invariance is easy to verify, because only the *f*-ranking is used in the algorithm, which is invariant under the choice of *g*.

- Scale-invariance, in that for any $h : \mathbb{R}^n \to \mathbb{R}$ the behavior is independent of $\alpha > 0$ for the objective function $f : x \mapsto h(\alpha x)$ given $\sigma_0 \propto 1/\alpha$ and $m_0 \propto 1/\alpha$.

- Invariance under rotation of the search space in that for any $h : \mathbb{R}^n \to \mathbb{R}$ and any $z \in \mathbb{R}^n$ the behavior on $f : x \mapsto h(Rx)$ is independent of the orthogonal matrix *R*, given $m_0 = R^{-1}z$. More general, the algorithm is also invariant under general linear transformations *R* when additionally the initial covariance matrix is chosen as $R^{-1}R^{-1^T}$.

Any serious parameter optimization method should be translation invariant, but most methods do not exhibit all the above described invariance properties. A prominent example with the same invariance properties is the Nelder–Mead method, where the initial simplex must be chosen respectively.

Convergence

Conceptual considerations like the scale-invariance property of the algorithm, the analysis of simpler evolution strategies, and overwhelming empirical evidence suggest that the algorithm converges on a large class of functions fast to the global optimum, denoted as x^*. On some functions, convergence occurs independently of the initial conditions with probability one. On some functions the probability is smaller than one and typically depends on the initial m_0 and σ_0. Empirically, the fastest possible convergence rate in *k* for rank-based direct search methods can often be observed (depending on the context denoted as *linear* or *log-linear* or *exponential* convergence). Informally, we can write

$$\|m_k - x^*\| \approx \|m_0 - x^*\| \times e^{-ck}$$

for some $c > 0$, and more rigorously

$$\frac{1}{k}\sum_{i=1}^{k} \log \frac{\|m_i - x^*\|}{\|m_{i-1} - x^*\|} = \frac{1}{k} \log \frac{\|m_k - x^*\|}{\|m_0 - x^*\|} \to -c < 0 \quad \text{for } k \to \infty\,,$$

or similarly,

$$E \log \frac{\|m_k - x^*\|}{\|m_{k-1} - x^*\|} \to -c < 0 \quad \text{for } k \to \infty\,.$$

This means that on average the distance to the optimum is decreased in each iteration by a "constant" factor, namely by $\exp(-c)$. The convergence rate c is roughly $0.1\lambda / n$, given λ is not much larger than the dimension. In any case, c cannot exceed $0.25\lambda / n$, given the above recombination weights w_i are all non-negative. The linear dependencies in λ and n are however remarkable and they are in both cases the best one can hope for in this kind of algorithm. Yet, a rigorous proof of convergence is missing.

Interpretation as Coordinate System Transformation

Using a non-identity covariance matrix for the multivariate normal distribution in evolution strategies is equivalent to a coordinate system transformation of the solution vectors, mainly because the sampling equation

$$\begin{aligned} x_i &\sim m_k + \sigma_k \times \mathcal{N}(0, C_k) \\ &\sim m_k + \sigma_k \times C_k^{1/2} \mathcal{N}(0, I) \end{aligned}$$

can be equivalently expressed in an "encoded space" as

$$\underbrace{C_k^{-1/2} x_i}_{\text{represented in the encode space}} \sim \underbrace{C_k^{-1/2} m_k}_{\text{represented in the encode space}} + \sigma_k \times \mathcal{N}(0, I)$$

The covariance matrix defines a bijective transformation (encoding) for all solution vectors into a space, where the sampling takes place with identity covariance matrix. Because the update equations in the CMA-ES are invariant under coordinate system transformations (general linear transformations), the CMA-ES can be re-written as an adaptive encoding procedure applied to a simple evolution strategy with identity covariance matrix. This adaptive encoding procedure is not confined to algorithms that sample from a multivariate normal distribution (like evolution strategies), but can in principle be applied to any iterative search method.

Performance in Practice

In contrast to most other evolutionary algorithms, the CMA-ES is quasi parameter-free. It has been empirically successful in hundreds of applications and is considered to be useful in particular on non-convex, non-separable, ill-conditioned, multi-modal or noisy objective functions. The search space dimension ranges typically between two and a few hundred. Assuming a black-box optimization scenario, where gradients are not available (or not useful) and function evaluations are the only considered cost of search, the CMA-ES method is likely to be outperformed by other methods in the following conditions.

- on low-dimensional functions, say $n < 5$, for example by the downhill simplex method or surrogate-based methods (like kriging with expected improvement)
- on separable functions without or with only negligible dependencies between the design variables in particular in the case of multi-modality or large dimension, for example by differential evolution
- on (nearly) convex-quadratic functions with low or moderate condition number of the Hessian matrix, where BFGS or NEWUOA are typically ten times faster
- on functions that can already be solved with a comparatively small number of function evaluations, say no more than $10n$, where CMA-ES is often slower than, for example, NEWUOA or Multilevel Coordinate Search (MCS).

On separable functions the performance disadvantage is likely to be most significant, in that CMA-ES might not be able to find at all comparable solutions. On the other hand, on non-separable functions that are ill-conditioned or rugged or can only be solved with more than $100n$ function evaluations, the CMA-ES shows most often superior performance.

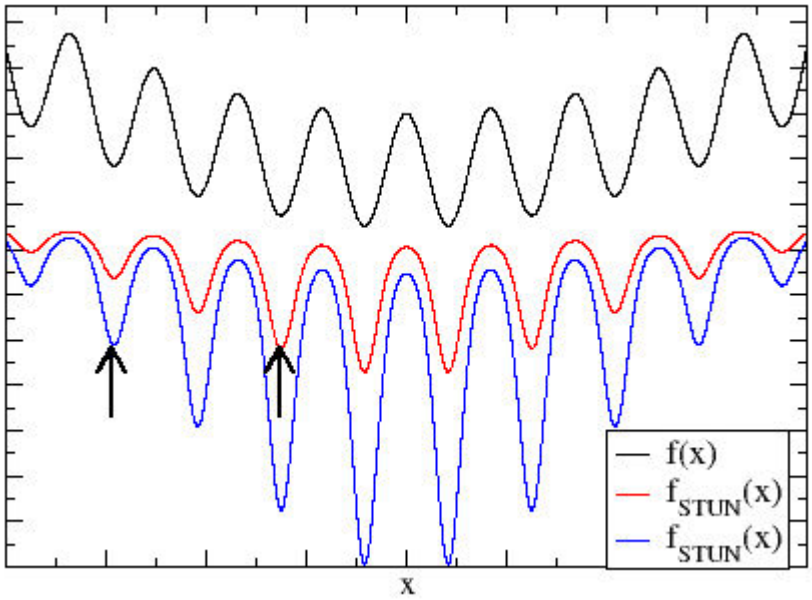
Variations and Extensions

The (1+1)-CMA-ES generates only one candidate solution per iteration step which only becomes the new distribution mean, if it is better than the old mean. For $c_c = 1$ it is a close variant of Gaussian adaptation. The CMA-ES has also been extended to multiobjective optimization as MO-CMA-ES . Another remarkable extension has been the addition of a negative update of the covariance matrix with the so-called active CMA.

Stochastic tunneling

Stochastic tunneling (STUN) is an approach to global optimization based on the Monte Carlo method-sampling of the function to be minimized.

Idea



Schematic one-dimensional test function (black) and STUN effective potential (red & blue), where the minimum indicated by the arrows is the best minimum found so far. All wells that lie above the best minimum found are suppressed. If the dynamical process can escape the well around the current minimum estimate it will not be trapped by other local minima that are higher. Wells with deeper minima are enhanced. The dynamical process is accelerated by that.

Monte Carlo method-based optimization techniques sample the objective function by randomly "hopping" from the current solution vector to another with a difference in the function value of ΔE. The acceptance probability of such a trial jump is in most cases chosen to be $\min\left(1; \exp\left(-\beta \cdot \Delta E\right)\right)$ (Metropolis criterion) with an appropriate parameter β.

The general idea of STUN is to circumvent the slow dynamics of ill-shaped energy functions that one encounters for example in spin glasses by tunneling through such barriers.

This goal is achieved by Monte Carlo sampling of a transformed function that lacks this slow dynamics. In the "standard-form" the transformation reads $f_{STUN} := 1 - \exp\left(-\gamma \cdot \left(f(x) - f_o\right)\right)$ where f_o is the lowest function value found so far. This transformation preserves the loci of the minima.

The effect of such a transformation is shown in the graph.

Chapter- 5

Particle Swarm Optimization

In computer science, **particle swarm optimization** (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. Such methods are commonly known as metaheuristics as they make few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics such as PSO do not guarantee an optimal solution is ever found.

More specifically, PSO does not use the gradient of the problem being optimized, which means PSO does not require for the optimization problem to be differentiable as is required by classic optimization methods such as gradient descent and quasi-newton methods. PSO can therefore also be used on optimization problems that are partially irregular, noisy, change over time, etc.

PSO optimizes a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae. The movements of the particles are guided by the best found positions in the search-space which are updated as better positions are found by the particles.

PSO is originally attributed to Kennedy, Eberhart and Shi and was first intended for simulating social behaviour . The algorithm was simplified and it was observed to be performing optimization. The book by Kennedy and Eberhart describes many philosophical aspects of PSO and swarm intelligence. An extensive survey of PSO applications is made by Poli.

Algorithm

A basic variant of the PSO algorithm works by having a population (called a swarm) of candidate solutions (called particles). These particles are moved around in the search-

space according to a few simple formulae. The movements of the particles are guided by their own best known position in the search-space as well as the entire swarm's best known position. When improved positions are being discovered these will then come to guide the movements of the swarm. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered.

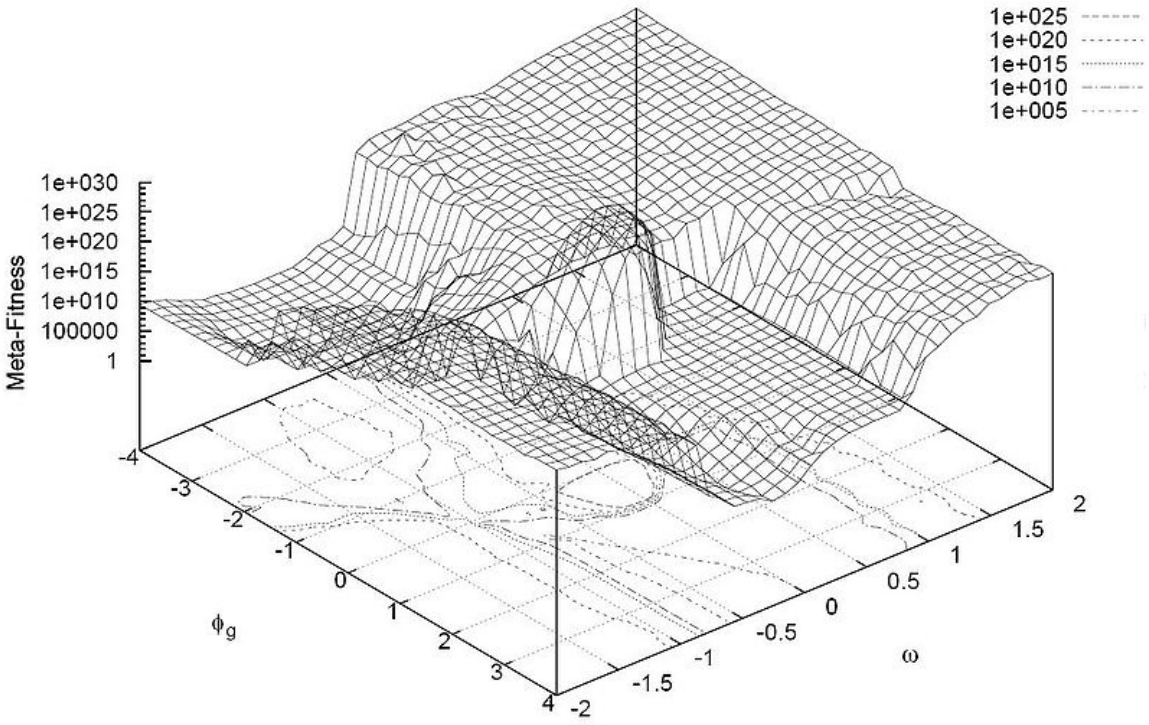
Formally, let $f: \mathbb{R}^n \to \mathbb{R}$ be the fitness or cost function which must be minimized. The function takes a candidate solution as argument in the form of a vector of real numbers and produces a real number as output which indicates the fitness of the given candidate solution. The gradient of f is not known. The goal is to find a solution $\mathbf{a}$ for which $f(\mathbf{a}) \leq f(\mathbf{b})$ for all $\mathbf{b}$ in the search-space, which would mean $\mathbf{a}$ is the global minimum. Maximization can be performed by considering the function $h = -f$ instead.

Let S be the number of particles in the swarm, each having a position $\mathbf{x}_i \in \mathbb{R}^n$ in the search-space and a velocity $\mathbf{v}_i \in \mathbb{R}^n$. Let $\mathbf{p}_i$ be the best known position of particle i and let $\mathbf{g}$ be the best known position of the entire swarm. A basic PSO algorithm is then:

- For each particle $i = 1, ..., S$ do:
  - Initialize the particle's position with a uniformly distributed random vector: $\mathbf{x}_i \sim U(\mathbf{b_{lo}}, \mathbf{b_{up}})$, where $\mathbf{b_{lo}}$ and $\mathbf{b_{up}}$ are the lower and upper boundaries of the search-space.
  - Initialize the particle's best known position to its initial position: $\mathbf{p}_i \leftarrow \mathbf{x}_i$
  - If $(f(\mathbf{p}_i) < f(\mathbf{g}))$ update the swarm's best known position: $\mathbf{g} \leftarrow \mathbf{p}_i$
  - Initialize the particle's velocity: $\mathbf{v}_i \sim U(-|\mathbf{b_{up}}-\mathbf{b_{lo}}|, |\mathbf{b_{up}}-\mathbf{b_{lo}}|)$
- Until a termination criterion is met (e.g. number of iterations performed, or adequate fitness reached), repeat:
  - For each particle $i = 1, ..., S$ do:
    - Pick random numbers: $r_p, r_g \sim U(0,1)$
    - Update the particle's velocity: $\mathbf{v}_i \leftarrow \omega\, \mathbf{v}_i + \varphi_p\, r_p\, (\mathbf{p}_i-\mathbf{x}_i) + \varphi_g\, r_g\, (\mathbf{g}-\mathbf{x}_i)$
    - Update the particle's position: $\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$
    - If $(f(\mathbf{x}_i) < f(\mathbf{p}_i))$ do:
      - Update the particle's best known position: $\mathbf{p}_i \leftarrow \mathbf{x}_i$
      - If $(f(\mathbf{p}_i) < f(\mathbf{g}))$ update the swarm's best known position: $\mathbf{g} \leftarrow \mathbf{p}_i$
- Now $\mathbf{g}$ holds the best found solution.

The parameters ω, φ_p, and φ_g are selected by the practitioner and control the behaviour and efficacy of the PSO method, see below.

Parameter selection



Performance landscape showing how a simple PSO variant performs in aggregate on several benchmark problems when varying two PSO parameters.

The choice of PSO parameters can have a large impact on optimization performance. Selecting PSO parameters that yield good performance has therefore been the subject of much research, e.g. Shi and Eberhart , Carlisle and Dozier , van den Bergh , Clerc and Kennedy , Trelea , Bratton and Blackwell , and Evers .

A simple and efficient way of tuning the PSO parameters was presented by Pedersen et al. who also made numerous experiments with different optimization problems and settings. The technique for tuning PSO parameters is called meta-optimization because another optimization method is used in an overlaying manner to tune the PSO parameters. Parameters tuned for various optimization scenarios are listed in.

Inner Workings

There are several schools of thought as to why and how the PSO algorithm can perform optimization.

A common belief amongst researchers is that the swarm behaviour varies between exploratory behaviour, that is, searching a broader region of the search-space, and exploitative behaviour, that is, a locally oriented search so as to get closer to a (possibly

local) optimum. This school of thought has been prevalent since the inception of PSO, for example Kennedy , Shi and Eberhart , and Clerc and Kennedy . This school of thought contends that the PSO algorithm and its parameters must be chosen so as to properly balance between exploration and exploitation to avoid premature convergence to a local optimum yet still ensure a good rate of convergence to the optimum. This belief is the precursor of many PSO variants, see below.

Another school of thought is that the behaviour of a PSO swarm is not well understood in terms of how it affects actual optimization performance, especially for higher dimensional search-spaces and optimization problems that may be discontinuous, noisy, and time-varying. This school of thought merely tries to find PSO algorithms and parameters that cause good performance regardless of how the swarm behaviour can be interpreted in relation to e.g. exploration and exploitation. Such studies have led to the simplification of the PSO algorithm, see below.

Convergence

In relation to PSO the word *convergence* typically means one of two things, although it is often not clarified which definition is meant and sometimes they are mistakenly thought to be identical.

- Convergence may refer to the swarm's best known position **g** approaching (converging to) the optimum of the problem, regardless of how the swarm behaves.
- Convergence may refer to a swarm collapse in which all particles have converged to a point in the search-space, which may or may not be the optimum.

Several attempts at mathematically analyzing PSO convergence exist in the literature. These analyses have resulted in guidelines for selecting PSO parameters that are believed to cause convergence, divergence or oscillation of the swarm's particles, and the analyses have also given rise to several PSO variants. However, the analyses were criticized by Pedersen for being oversimplified as they assume the swarm has only one particle, that it does not use stochastic variables and that the points of attraction, that is, the particle's best known position **p** and the swarm's best known position **g**, remain constant throughout the optimization process. Furthermore, the elaborate analysis by e.g. van den Bergh (section 3.3) proves that certain PSO variants converge to the global optimum, however, the proof allows for an infinite number of optimization iterations to be performed and Pedersen notes that under such assumptions convergence to the optimum can in fact be proven trivially for completely random sampling of the search-space (section 1.4.4). Such claims of convergence are therefore of no practical use without an estimate on how many iterations the optimizer will need to discover a satisfactory solution. Determining convergence capabilities of different PSO algorithms and parameters therefore still depends on empirical results.

Variants

New PSO variants are continually being introduced in an attempt to improve optimization performance. There are certain trends in that research, one is to make a hybrid optimization method using PSO combined with other optimizers, e.g., another research trend is to try and alleviate premature convergence (that is, optimization stagnation) e.g. by reversing or perturbing the movement of the PSO particles, e.g., and then there are also attempts at adapting the behavioural parameters of PSO during optimization, e.g..

Simplifications

Another school of thought is that PSO should be simplified as much as possible without impairing its performance; a general concept often referred to as Occam's razor. Simplifying PSO was originally suggested by Kennedy and studied more extensively by Bratton and Blackwell , as well as Pedersen and Chipperfield where it appeared that optimization performance was improved, and the parameters were easier to tune and they performed more consistently across different optimization problems.

Another argument in favour of simplifying PSO is that metaheuristics can only have their efficacy demonstrated empirically by doing computational experiments on a finite number of optimization problems. This means a metaheuristic such as PSO cannot be proven correct and this increases the risk of making errors in its description and implementation. A good example of this can be found in which presented a promising variant of a genetic algorithm (another popular metaheuristic) but it was later found to be defective as it was strongly biased in its optimization search towards the origin which happened to be the optimum of the benchmark problems considered .

Chapter- 6

Bees and Firefly Algorithm

Bees algorithm

In computer science and operations research, the **bees algorithm** is a population-based search algorithm first developed in 2005. It mimics the food foraging behaviour of swarms of honey bees. In its basic version, the algorithm performs a kind of neighbourhood search combined with random search and can be used for both combinatorial optimization and functional optimisation.

The foraging process in nature

A colony of honey bees can extend itself over long distances (up to 14 km) and in multiple directions simultaneously to exploit a large number of food sources. A colony prospers by deploying its foragers to good fields. In principle, flower patches with plentiful amounts of nectar or pollen that can be collected with less effort should be visited by more bees, whereas patches with less nectar or pollen should receive fewer bees.

The foraging process begins in a colony by scout bees being sent to search for promising flower patches. Scout bees move randomly from one patch to another. During the harvesting season, a colony continues its exploration, keeping a percentage of the population as scout bees.

When they return to the hive, those scout bees that found a patch which is rated above a certain quality threshold (measured as a combination of some constituents, such as sugar content) deposit their nectar or pollen and go to the “dance floor” to perform a dance known as the waggle dance.

This dance is essential for colony communication, and contains three pieces of information regarding a flower patch: the direction in which it will be found, its distance from the hive and its quality rating (or fitness). This information helps the colony to send

its bees to flower patches precisely, without using guides or maps. Each individual's knowledge of the outside environment is gleaned solely from the waggle dance. This dance enables the colony to evaluate the relative merit of different patches according to both the quality of the food they provide and the amount of energy needed to harvest it. After waggle dancing inside the hive, the dancer (i.e. the scout bee) goes back to the flower patch with follower bees that were waiting inside the hive. More follower bees are sent to more promising patches. This allows the colony to gather food quickly and efficiently.

While harvesting from a patch, the bees monitor its food level. This is necessary to decide upon the next waggle dance when they return to the hive. If the patch is still good enough as a food source, then it will be advertised in the waggle dance and more bees will be recruited to that source.

The Bees Algorithm

The Bees Algorithm is an optimisation algorithm inspired by the natural foraging behaviour of honey bees to find the optimal solution. The algorithm requires a number of parameters to be set, namely: number of scout bees (n), number of sites selected out of n visited sites (m), number of best sites out of m selected sites (e), number of bees recruited for best e sites (nep), number of bees recruited for the other (m-e) selected sites (nsp), initial size of patches (ngh) which includes site and its neighbourhood and stopping criterion.

The pseudo code for the bees algorithm in its simplest form:

```
  1. Initialise population with random solutions.
  2. Evaluate fitness of the population.
  3. While (stopping criterion not met) //Forming new population.
  4.   Select sites for neighbourhood search.
  5.   Recruit bees for selected sites (more bees for best e sites) and
evaluate fitnesses.
  6.   Select the fittest bee from each patch.
  7.   Assign remaining bees to search randomly and evaluate their
fitnesses.
  8. End While.
```

In first step, the bees algorithm starts with the scout bees (n) being placed randomly in the search space. In step 2, the fitnesses of the sites visited by the scout bees are evaluated. In step 4, bees that have the highest fitnesses are chosen as "selected bees" and sites visited by them are chosen for neighbourhood search. Then, in steps 5 and 6, the algorithm conducts searches in the neighbourhood of the selected sites, assigning more bees to search near to the best e sites. The bees can be chosen directly according to the fitnesses associated with the sites they are visiting. Alternatively, the fitness values are used to determine the probability of the bees being selected. Searches in the neighbourhood of the best e sites which represent more promising solutions are made more detailed by recruiting more bees to follow them than the other selected bees.

Together with scouting, this differential recruitment is a key operation of the Bees Algorithm. However, in step 6, for each patch only the bee with the highest fitness will be selected to form the next bee population. In nature, there is no such a restriction. This restriction is introduced here to reduce the number of points to be explored. In step 7, the remaining bees in the population are assigned randomly around the search space scouting for new potential solutions. These steps are repeated until a stopping criterion is met. At the end of each iteration, the colony will have two parts to its new population – those that were the fittest representatives from a patch and those that have been sent out randomly.

Applications

The Bees Algorithm, which is inspired by the food foraging behaviour of honey bees, has found many applications in engineering field, such as:

- Training neural networks for pattern recognition.
- Forming manufacturing cells.
- Scheduling jobs for a production machine.
- Solving continuous problems and engineering optimization.
- Finding multiple feasible solutions to a preliminary design problems.
- Data clustering
- Optimising the design of mechanical components.
- Multi-Objective Optimisation.
- Tuning a fuzzy logic controller for a robot gymnast.
- Computer Vision and Image Analysis.

Firefly algorithm

The **firefly algorithm (FA)** is a metaheuristic algorithm, inspired by the flashing behaviour of fireflies. The primary purpose for a firefly's flash is to act as a signal system to attract other fireflies. Xin-She Yang formulated this firefly algorithm by assuming:

1. All fireflies are unisex, so that one firefly will be attracted to all other fireflies;
2. Attractiveness is proportional to their brightness, and for any two fireflies, the less brighter one will attract (and thus move) to the brighter one; however, the brightness can decrease as their distance increases;
3. If there are no fireflies brighter than a given firefly, it will move randomly.

The brightness should be associated with the objective function.

Firefly algorithm is a nature-inspired metaheuristic optimization algorithm.

Algorithm description

The pseudo code can be summarized as:

Begin

1) Objective function: $f(\mathbf{x}), \quad \mathbf{x} = (x_1, x_2, ..., x_d)$;
2) Generate an initial population of fireflies $\mathbf{x}_i \quad (i = 1, 2, \ldots, n)$;.
3) Formulate light intensity *I* so that it is associated with $f(\mathbf{x})$
   (for example, for maximization problems, $I \propto f(\mathbf{x})$ or simply $I = f(\mathbf{x})$;
4) Define absorption coefficient γ
While (t<MaxGeneration)
   for i=1:n (all n fireflies);
      for j=1:n (n fireflies)
         if (I_j > I_i),
          move firefly i towards j;
         end if
      Vary attractiveness with distance r via $\exp(-\gamma\ r^2)$;
      Evaluate new solutions and update light intensity;
      end for j
   end for i
   Rank fireflies and find the current best;
end while
Post-processing the results and visualization;

end

It can be shown that the limiting case $\gamma \to 0$corresponds to the standard Particle Swarm Optimization (PSO). In fact, if the inner loop (for j) is removed and the brightness I_j is replaced by the current global best g^*, then FA essentially becomes the standard PSO. In addition, the γ should be related to the scales of design variables. Parametric studies show that n (number of fireflies) should be about 15 to 40 for most problems.

A simple demo Matlab code is available matlab code'

Recent studies shows that the firefly algorithm is very efficient, and could outperform other metaheuristic algorithms including particle swarm optimization. Most metaheuristic algorithms may have difficulty in dealing with stochastic test functions, and it seems that firefly algorithm can deal with stochastic test functions very efficiently.

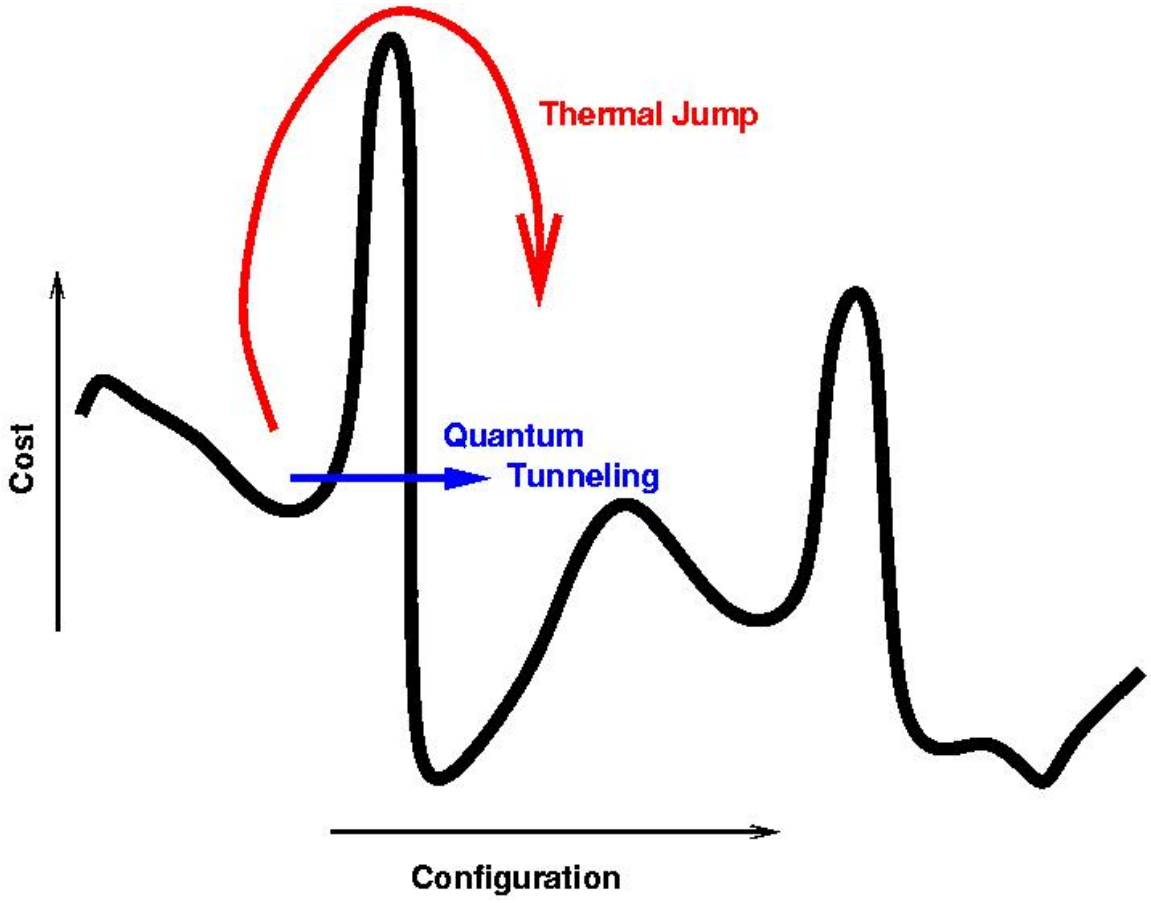
Discrete Firefly Algorithm (DFA)

A discrete version of Firefly Algorithm, namely, Discrete Firefly Algorithm (DFA) proposed recently by M. K. Sayadi, R. Ramezanian and N. Ghaffari-Nasab can efficiently solve NP-hard scheduling problems. DFA outperforms existing algorithms such as the ant colony algorithm.

Chapter- 7

Other Concepts in Optimization algorithms

Quantum annealing



In mathematics and applications, **quantum annealing** (QA) is a general method for finding the global minimum of a given *objective function* over a given set of *candidate*

solutions (the *search space*), by a process analogous to quantum fluctuations. It is used mainly for problems where the search space is discrete (combinatorial optimization problems) with many local minima; such as finding the ground states of a glassy system.

In quantum annealing, a "current state" (the current candidate solution) *s* is randomly replaced by a randomly selected *neighbor state s*' if the latter has a lower "energy" (value of the objective function). The process is controlled by the "tunneling field strength"*T*, a parameter that determines the extent of the neighborhood of states explored by the method. The tunneling field starts high, so that the neighborhood extends over the whole search space; and is slowly reduced through the computation, until the neighborhood shrinks to those few states that differ minimally from the current states.

Comparison to simulated annealing

Quantum annealing can be compared to simulated annealing (SA), whose "temperature" parameter plays a similar role to QA's tunneling field strength. However, in SA the neighborhood stays the same throughout the search, and the temperature determines the probability of moving to a state of higher "energy". In QA, the tunneling field strength determines instead the neighborhood radius, i.e. the mean distance between the next candidate *s*' and the current candidate *s*.

In more elaborated SA variants (such as Adaptive simulated annealing), the neighborhood radius is also varied using acceptance rate percentages or the temperature value.

Quantum mechanics analogy

The tunneling field is basically a kinetic energy term that does not commute with the classical potential energy part of the original glass. The whole process can be simulated in a computer using quantum Monte Carlo (or other stochastic technique), and thus obtain a heuristic algorithm for finding the ground state of the classical glass. It is speculated that in a quantum computer, such simulations would be much more efficient and exact than that done in a classical computer, due to *quantum parallelism* realized by the actual superposition of all the classical configurations at any instant.

By that time, the system finds a very deep (likely, the global one) minimum and settle there. At the end, we are left with the classical system at its global minimum.

In the case of annealing a purely mathematical *objective function*, one may consider the variables in the problem to be classical degrees of freedom, and the cost functions to be the potential energy function (classical Hamiltonian). Then a suitable term consisting of non-commuting variable(s) (i.e. variables that has non-zero commutator with the variables of the original mathematical problem) has to be introduced artificially in the Hamiltonian to play the role of the tunneling field (kinetic part). Then one may carry out the simulation with the quantum Hamiltonian thus constructed (the original function +

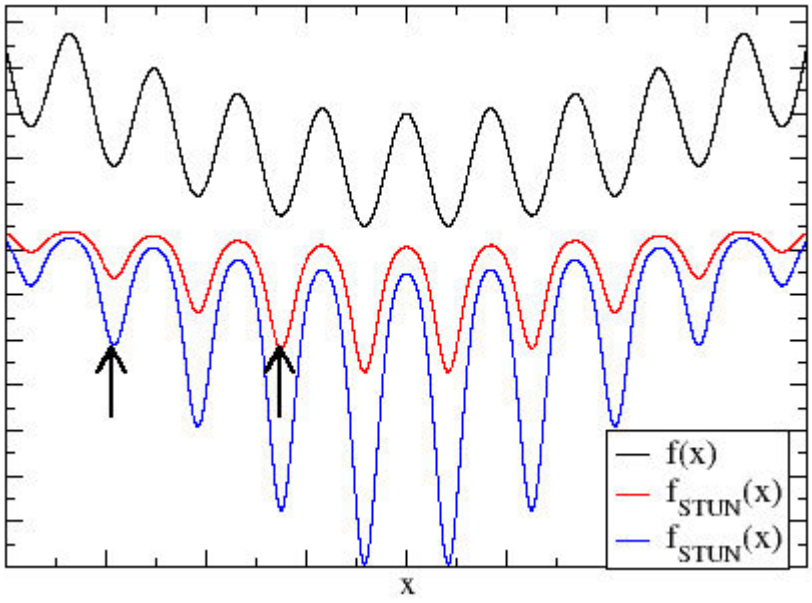
non-commuting part) just as described above. Here, there is a choice in selecting the non-commuting term and the efficiency of annealing may depend on that.

It has been demonstrated experimentally as well as theoretically, that quantum annealing can indeed out rate thermal annealing in certain cases, especially, where the potential energy (cost) landscape consists of very high but thin barriers surrounding shallow local minima. Since thermal transition probabilities (~exp($-\Delta / k_B T$); T => Temperature, k_B => Boltzmann constant) depend only on the height Δ of the barriers, it is very difficult for thermal fluctuations to get the system out from such local minima. But quantum tunneling probabilities through a barrier depend not only the height Δ of the barrier, but also on its width w; if the barriers are thin enough, quantum fluctuations may bring the system out of the shallow local minima surrounded by them.

Stochastic tunneling

Stochastic tunneling (STUN) is an approach to global optimization based on the Monte Carlo method-sampling of the function to be minimized.

Idea



Schematic one-dimensional test function (black) and STUN effective potential (red & blue), where the minimum indicated by the arrows is the best minimum found so far. All wells that lie above the best minimum found are suppressed. If the dynamical process can escape the well around the current minimum estimate it will not be trapped by other local minima that are higher. Wells with deeper minima are enhanced. The dynamical process is accelerated by that.

Monte Carlo method-based optimization techniques sample the objective function by randomly "hopping" from the current solution vector to another with a difference in the function value of ΔE. The acceptance probability of such a trial jump is in most cases chosen to be $\min\left(1; \exp\left(-\beta \cdot \Delta E\right)\right)$(Metropolis criterion) with an appropriate parameter β.

The general idea of STUN is to circumvent the slow dynamics of ill-shaped energy functions that one encounters for example in spin glasses by tunneling through such barriers.

This goal is achieved by Monte Carlo sampling of a transformed function that lacks this slow dynamics. In the "standard-form" the transformation reads $f_{STUN} := 1 - \exp\left(-\gamma \cdot \left(f(x) - f_o\right)\right)$where f_o is the lowest function value found so far. This transformation preserves the loci of the minima.

The effect of such a transformation is shown in the graph.

Guided Local Search

Guided Local Search is a metaheuristic search method. A meta-heuristic method is a method that sits on top of a local search algorithm to change its behaviour.

Guided Local Search builds up penalties during a search. It uses penalties to help local search algorithms escape from local minima and plateaus. When the given local search algorithm settles in a local optimum, GLS modifies the objective function using a specific scheme (explained below). Then the local search will operate using an augmented objective function, which is designed to bring the search out of the local optimum. The key is in the way that the objective function is modified.

Overview

Solution features

To apply GLS, solution features must be defined for the given problem. Solution features are defined to distinguish between solutions with different characteristics, so that regions of similarity around local optima can be recognized and avoided. The choice of solution features depends on the type of problem, and also to a certain extent on the local search algorithm. For each feature f_i a cost function c_i is defined.

Each feature is also associated with a penalty p_i (initially set to 0) to record the number of occurrences of the feature in local minima.

The features and costs often come directly from the objective function. For example, in the traveling salesman problem, “whether the tour goes directly from city X to city Y”

can be defined to be a feature. The distance between X and Y can be defined to be the cost. In the SAT and weighted MAX-SAT problems, the features can be "whether clause C satisfied by the current assignments".

At the implementation level, we define for each feature *i* an Indicator Function I_i indicating whether the feature is present in the current solution or not. I_i is 1 when solution *x* exhibits property *i*, 0 otherwise.

Selective penalty modifications

GLS computes the utility of penalising each feature. When the Local Search algorithm returns a local minimum x, GLS penalizes all those features (through increments to the penalty of the features) present in that solution which have maximum utility, $\text{util}(x, i)$, as defined below.

$$\text{util}(x,i) = I_i(x)\frac{c_i(x)}{1+p_i}.$$

The idea is to penalise features that have high costs, although the utility of doing so decreases as the feature is penalised more and more often.

Searching through an augmented cost function

GLS uses an augmented cost function (defined below), to allow it to guide the Local Search algorithm out of the local minimum, through penalising features present in that local minimum. The idea is to make the local minimum more costly than the surrounding search space, where these features are not present.

$$g(x) = f(x) + \lambda a \sum_{1 \le i \le m} I_i(x) p_i$$

The parameter λ may be used to alter the intensification of the search for solutions. A higher value for λ will result in a more diverse search, where plateaus and basins are searched more coarsely; a low value will result in a more intensive search for the solution, where the plateaus and basins in the search landscape are searched in finer detail. The coefficient *a* is used to make the penalty part of the objective function balanced relative to changes in the objective function and is problem specific. A simple heuristic for setting *a* is simply to record the average change in objective function up until the first local minimum, and then set *a* to this value divided by the number of GLS features in the problem instance.

Extensions of Guided Local Search

Mills (2002) has described an Extended Guided Local Search (EGLS) which utilises random moves and an aspiration criterion designed specifically for penalty based

schemes. The resulting algorithm improved the robustness of GLS over a range of parameter settings, particularly in the case of the quadratic assignment problem. A general version of the GLS algorithm, using a min-conflicts based hill climber (Minton et al. 1992) and based partly on GENET for constraint satisfaction and optimisation, has also been implemented in the Computer Aided Constraint Programming project.

Related work

GLS was built on GENET, which was developed by Chang Wang, Edward Tsang and Andrew Davenport.

The breakout method is very similar to GENET. It was designed for constraint satisfaction.

Tabu search is a class of search methods which can be instantiated to specific methods. GLS can be seen as a special case of Tabu search.

By sitting GLS on top of genetic algorithm, Tung-leng Lau introduced the Guided Genetic Programming (GGA) algorithm. It was successfully applied to the general assignment problem (in scheduling), processors configuration problem (in electronic design) and a set of radio-link frequency assignment problems (an abstracted military application).

Choi et al. cast GENET as a Lagrangian search.

Tabu search

Tabu search is a mathematical optimization method, belonging to the class of local search techniques. Tabu search enhances the performance of a local search method by using memory structures: once a potential solution has been determined, it is marked as "taboo" ("tabu" being a different spelling of the same word) so that the algorithm does not visit that possibility repeatedly. Tabu search is attributed to Fred W. Glover.

Basic details

Tabu search is a metaheuristic algorithm that can be used for solving combinatorial optimization problems, such as the traveling salesman problem (TSP). Tabu search uses a local or neighborhood search procedure to iteratively move from a solution x to a solution x' in the neighborhood of x, until some stopping criterion has been satisfied. To explore regions of the search space that would be left unexplored by the local search procedure, tabu search modifies the neighborhood structure of each solution as the search progresses. The solutions admitted to $N^*(x)$, the new neighborhood, are determined through the use of memory structures. The search then progresses by iteratively moving from a solution x to a solution x' in $N^*(x)$.

Perhaps the most important type of memory structure used to determine the solutions admitted to $N^*(x)$ is the tabu list. In its simplest form, a tabu list is a short-term memory which contains the solutions that have been visited in the recent past (less than n iterations ago, where n is the number of previous solutions to be stored (n is also called the tabu tenure)). Tabu search excludes solutions in the tabu list from $N^*(x)$. A variation of a tabu list prohibits solutions that have certain attributes (e.g., solutions to the traveling salesman problem (TSP) which include undesirable arcs) or prevent certain moves (e.g. an arc that was added to a TSP tour cannot be removed in the next n moves). Selected attributes in solutions recently visited are labeled "tabu-active." Solutions that contain tabu-active elements are “tabu”. This type of short-term memory is also called "recency-based" memory.

Tabu lists containing attributes can be more effective for some domains, although they raise a new problem. When a single attribute is marked as tabu, this typically results in more than one solution being tabu. Some of these solutions that must now be avoided could be of excellent quality and might not have been visited. To mitigate this problem, "aspiration criteria" are introduced: these override a solution's tabu state, thereby including the otherwise-excluded solution in the allowed set. A commonly used aspiration criterion is to allow solutions which are better than the currently-known best solution.

Example: Traveling salesman problem

The traveling salesman problem (TSP) is often used to show the functionality of tabu search. The TSP involves finding an ordering of travel between cities, such that the distance traveled is minimized. For example, if city A and city B are next to each other, while city C is farther away, the total distance traveled will be shorter if cities A and B are visited one after the other, before visiting city C. Since finding an optimal order of visiting cities in the TSP is NP-hard, heuristic-based approximation methods are useful for approaching optimality.

Tabu search can be used to find a satisficing solution for the TSP. First, tabu search starts with an initial solution, which can be generated according to the nearest neighbor algorithm. To create new solutions, the order that two cities are visited is swapped. The distance for the total travel between all the cities is used to judge how good one solution is over another. To prevent cycles and to get out of local optima, a solution is added to the tabu list if it is accepted into $N^*(x)$, the solution neighborhood. New solutions continue to be created until some stopping criteria, such as an arbitrary number of iterations, is met. Once tabu search stops, the best solution is the solution with the shortest distance for the total travel between all cities.

Stochastic gradient descent

Stochastic gradient descent is an optimization method for minimizing an objective function that is written as a sum of differentiable functions.

Background

Both statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum:

$$Q(w) = \sum_{i=1}^{n} Q_i(w),$$

where the parameter w is to be estimated and where typically each summand function $Q_i()$ is associated with the i-th observation in the data set (used for training).

In classical statistics, sum-minimization problems arise in least squares and of maximum-likelihood estimation (for independent observations). The general class of estimators that arise as minimizers of sums are called M-estimators. However, in statistics, it has been long recognized that requiring even local minimization is too restrictive for some problems of maximum-likelihood estimation, as shown for example by Thomas Ferguson's example. Therefore, contemporary statisticial theorists often consider stationary points of the likelihood function (or zeros of its derivative, the score function, and other estimating equations).

The sum-minimization problem also arises for empirical risk minimization: In this case, $Q_i(w)$ is the value of loss function at i-th example, and $Q(w)$ is the empirical risk.

When used to minimize the above function, a standard (or "batch") gradient descent method would perform the following iterations :

$$w := w - \alpha \nabla Q(w) = w - \alpha \sum_{i=1}^{n} \nabla Q_i(w),$$

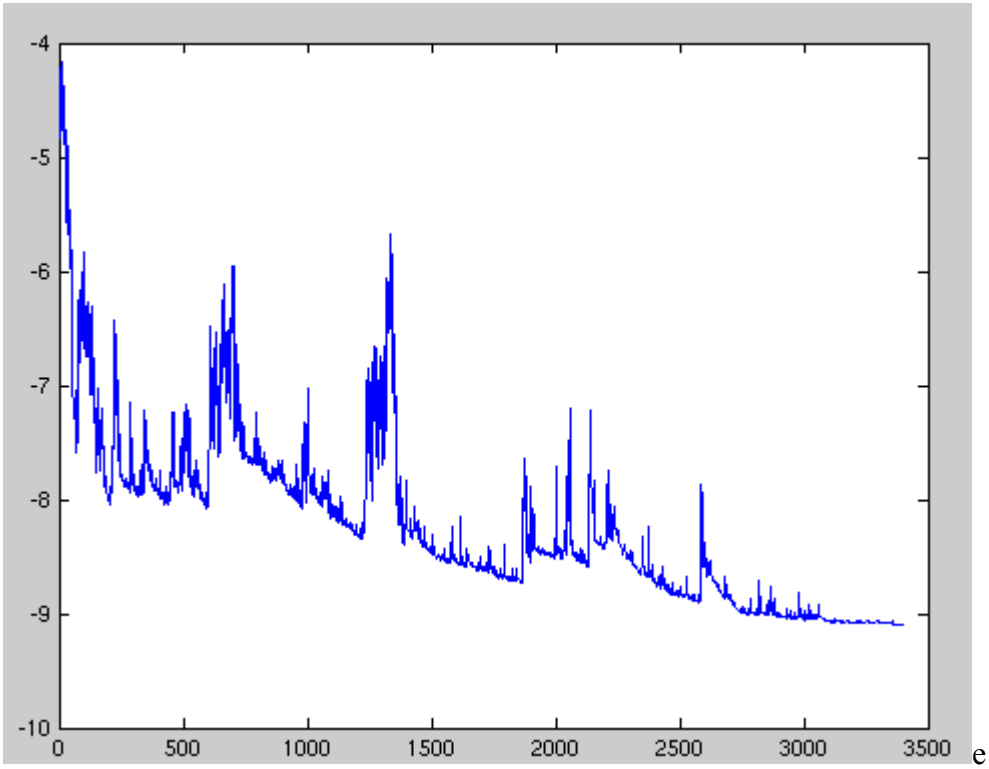
where α is a step size (sometimes called the *learning rate* in machine learning).

In many cases, the summand functions have a simple form that enables inexpensive evaluations of the sum-function and the sum gradient. For example, in statistics, one-parameter exponential families allow economical function-evaluations and gradient-evaluations.

However, in other cases, evaluating the sum-gradient may require expensive evaluations of the gradients from all summand functions. When the training set is enormous and no simple formulas exist, evaluating the sums of gradients becomes very expensive, because

evaluating the gradient requires evaluating all the summand functions' gradients. To economize on the computational cost at every iteration, stochastic gradient descent samples a subset of summand functions at every step.

Iterative method



e

Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

In stochastic (or "on-line") gradient descent, the true gradient of $Q(w)$ is approximated by a gradient at a single example:

$$w := w - \alpha \nabla Q_i(w).$$

As the algorithm sweeps through the training set, it performs the above update for each training example. Several passes over the training set are made until the algorithm converges. Typical implementations may also randomly shuffle training examples at each pass and use an adaptive learning rate.

In pseudocode, stochastic gradient descent with shuffling of training set at each pass can be presented as follows:

- Choose an initial vector of parameters w and learning rate α.
- Repeat until an approximate minimum is obtained:

- Randomly shuffle examples in the training set.
- For $i = 1, 2, ..., n$, do:
  - $w := w - \alpha \nabla Q_i(w).$

There is a compromise between the two forms, which is often called "mini-batches", where the true gradient is approximated by a sum over a small number of training examples.

The convergence of stochastic gradient descent has been analyzed using the theories of convex minimization and of stochastic approximation. Briefly, stochastic gradient methods need not converge to a global minimum unless the objective function is convex --- or more generally unless the problem has convex lower level sets and the function has the monotone property called "pseudoconvexity" in optimization theory, and finally the step-sizes are very small).

Example

Let's suppose we want to fit a straight line $y = w_1 + w_2 x$ to a training set of two-dimensional points $(x_1, y_1), \ldots, (x_n, y_n)$ using least squares. The objective function to be minimized is:

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{n} \left(w_1 + w_2 x_i - y_i \right)^2 .$$

The last line in the above pseudocode for this specific problem will become:

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} := \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} - \alpha 2 (w_1 + w_2 x_i - y_i) \begin{pmatrix} 1 \\ x_i \end{pmatrix}.$$

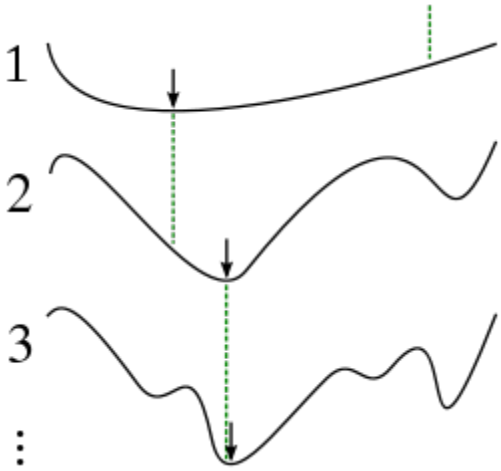
Applications

Some of the most popular stochastic gradient descent algorithms are the least mean squares (LMS) adaptive filter and the backpropagation algorithm.

Graduated optimization

Graduated optimization is a global optimization technique that attempts to solve a difficult optimization problem by initially solving a greatly simplified problem, and progressively transforming that problem (while optimizing) until it is equivalent to the difficult optimization problem.

Technique description



Graduated optimization is an improvement to hill climbing that enables a hill climber to avoid settling into local optima. It breaks a difficult optimization problem into a sequence of optimization problems, such that the first problem in the sequence is convex (or nearly convex), the solution to each problem gives a good starting point to the next problem in the sequence, and the last problem in the sequence is the difficult optimization problem that it ultimately seeks to solve. Often, graduated optimization gives better results than simple hill climbing. Further, when certain conditions exist, it can guarantees to find an optimal solution to the final problem in the sequence. These conditions are:

- The first optimization problem in the sequence can be solved given the initial starting point.
- The locally convex region around the global optimum of each problem in the sequence includes the point that corresponds to the global optimum of the previous problem in the sequence.

It can be shown inductively that if these conditions are met, then a hill climber will arrive at the global optimum for the difficult problem. Unfortunately, it can be difficult to find a sequence of optimization problems that meet these conditions. Often, graduated optimization yields good results even when the sequence of problems cannot be proven to strictly meet all of these conditions.

Some examples

Graduated optimization is commonly used in image processing for locating objects within a larger image. This problem can be made to be *more convex* by blurring the images. Thus, objects can be found by first searching the most-blurred image, then starting at that point and searching within a less-blurred image, and continuing in this manner until the object is located with precision in the original sharp image. (Representing images with varying levels of blurring is called pyramid representation, but

pyramid representation is also used for other purposes besides finding objects with graduated optimization.)

Graduated optimization can be used in manifold learning. The Manifold Sculpting algorithm, for example, uses graduated optimization to seek a manifold embedding for non-linear dimensionality reduction. It gradually scales variance out of extra dimensions within a data set while optimizing in the remaining dimensions. It, therefore, begins in a globally optimal state with a trivial problem, and as it scales variance out of the extra dimensions, it gradually transforms the problem into that of projecting data into a small number of dimensions.

It has also been used to calculate conditions for fractionation with tumors, for object tracking in computer vision, and other purposes.

Related optimization techniques

Simulated annealing is closely related to graduated optimization. Instead of smoothing the function over which it is optimizing, simulated annealing randomly perturbs the current solution by a decaying amount, which may have a similar effect.

Cross-entropy method

The **cross-entropy (CE) method** attributed to Reuven Rubinstein is a general Monte Carlo approach to combinatorial and continuous multi-extremal optimization and importance sampling. The method originated from the field of *rare event simulation*, where very small probabilities need to be accurately estimated, for example in network reliability analysis, queueing models, or performance analysis of telecommunication systems. The CE method can be applied to static and noisy combinatorial optimization problems such as the traveling salesman problem, the quadratic assignment problem, DNA sequence alignment, the max-cut problem and the buffer allocation problem, as well as continuous global optimization problems with many local extrema.

In a nutshell the CE method consists of two phases:

1. Generate a random data sample (trajectories, vectors, etc.) according to a specified mechanism.
2. Update the parameters of the random mechanism based on the data to produce a "better" sample in the next iteration. This step involves minimizing the *cross-entropy* or Kullback-Leibler divergence.

Estimation via importance sampling

Consider the general problem of estimating the quantity $\ell = \mathbb{E}_{\mathbf{u}}[H(\mathbf{X})] = \int H(\mathbf{x})\, f(\mathbf{x};\mathbf{u})\, \mathrm{d}\mathbf{x}$, where *H* is some *performance function* and $f(\mathbf{x};\mathbf{u})$ is a member of some parametric family of distributions. Using importance sampling this quantity can be estimated as $\hat{\ell} = \frac{1}{N}\sum_{i=1}^{N} H(\mathbf{X}_i)\frac{f(\mathbf{X}_i;\mathbf{u})}{g(\mathbf{X}_i)}$, where $\mathbf{X}_1, \ldots, \mathbf{X}_N$ is a random sample from g. For positive *H*, the theoretically *optimal* importance sampling density (pdf) is given by $g^*(\mathbf{x}) = H(\mathbf{x})f(\mathbf{x};\mathbf{u})/\ell$. This, however, depends on the unknown ℓ. The CE method aims to approximate the optimal pdf by adaptively selecting members of the parametric family that are closest (in the Kullback-Leibler sense) to the optimal pdf g^*.

Generic CE algorithm

1. Choose initial parameter vector $\mathbf{v}^{(0)}$; set t = 1.
2. Generate a random sample $\mathbf{X}_1, \ldots, \mathbf{X}_N$ from $f(\cdot;\mathbf{v}^{(t-1)})$
3. Solve for $\mathbf{v}^{(t)}$, where
   $$\mathbf{v}^{(t)} = \operatorname{argmax}_{\mathbf{v}} \frac{1}{N}\sum_{i=1}^{N} H(\mathbf{X}_i)\frac{f(\mathbf{X}_i;\mathbf{u})}{f(\mathbf{X}_i;\mathbf{v}^{(t-1)})}\log f(\mathbf{X}_i;\mathbf{v})$$
4. If convergence is reached then **stop**; otherwise, increase t by 1 and reiterate from step 2.

In several cases, the solution to step 3 can be found *analytically*. Situations in which this occurs are

- When f belongs to the natural exponential family
- When f is discrete with finite support
- When $H(\mathbf{X}) = \mathrm{I}_{\{\mathbf{x}\in A\}}$ and $f(\mathbf{X}_i;\mathbf{u}) = f(\mathbf{X}_i;\mathbf{v}^{(t-1)})$, then $\mathbf{v}^{(t)}$ corresponds to the maximum likelihood estimator based on those $\mathbf{X}_k \in A$.

Continuous optimization—example

The same CE algorithm can be used for optimization, rather than estimation. Suppose the problem is to maximize some function *S*(*x*), for example, $S(x) = \mathrm{e}^{-(x-2)^2} + 0.8\,\mathrm{e}^{-(x+2)^2}$. To apply CE, one considers first the *associated stochastic problem* of estimating $\mathbb{P}_{\boldsymbol{\theta}}(S(X) \geq \gamma)$ for a given *level* γ, and parametric

family $\{f(\cdot;\boldsymbol{\theta})\}$, for example the 1-dimensional Gaussian distribution, parameterized by its mean μ_t and variance σ_t^2 (so $\boldsymbol{\theta} = (\mu, \sigma^2)$ here). Hence, for a given γ, the goal is to find $\boldsymbol{\theta}$ so that $D_{\mathrm{KL}}(\mathrm{I}_{\{S(x)\geq\gamma\}} \| f_{\boldsymbol{\theta}})$ is minimized. This is done by solving the sample version (stochastic counterpart) of the KL divergence minimization problem, as in step 3 above. It turns out that parameters that minimize the stochastic counterpart for this choice of target distribution and parametric family are the sample mean and sample variance corresponding to the *elite samples*, which are those samples that have objective function value $\geq \gamma$. The worst of the elite samples is then used as the level parameter for the next iteration. This yields the following randomized algorithm for this problem.

Pseudo-code

```
1. mu:=-6; sigma2:=100; t:=0; maxits=100;    // Initialize parameters
2. N:=100; Ne:=10;                           //
3. while t < maxits and sigma2 > epsilon     // While not converged and
maxits not exceeded
4.  X = SampleGaussian(mu,sigma2,N);         // Obtain N samples from
current sampling distribution
5.  S = exp(-(X-2)^2) + 0.8 exp(-(X+2)^2);   // Evaluate objective
function at sampled points
6.  X = sort(X,S);                           // Sort X by objective
function values (in descending order)
7.  mu = mean(X(1:Ne)); sigma2=var(X(1:Ne)); // Update parameters of
sampling distribution
8.  t = t+1;                                 // Increment iteration
counter
9. return mu                                 // Return mean of final
sampling distribution as solution
```

Harmony search

In computer science and operations research, **harmony search** (HS) is a phenomenon-mimicking algorithm (also known as metaheuristic algorithm, soft computing algorithm or evolutionary algorithm) inspired by the improvisation process of musicians. In the HS algorithm, each musician (= decision variable) plays (= generates) a note (= a value) for finding a best harmony (= global optimum) all together. The Harmony Search algorithm has the following merits:

- HS does not require differential gradients, thus it can consider discontinuous functions as well as continuous functions.
- HS can handle discrete variables as well as continuous variables.
- HS does not require initial value setting for the variables.
- HS is free from divergence.
- HS may escape local optima.
- HS may overcome the drawback of GA's building block theory which works well only if the relationship among variables in a chromosome is carefully considered.

If neighbor variables in a chromosome have weaker relationship than remote variables, building block theory may not work well because of crossover operation. However, HS explicitly considers the relationship using ensemble operation.

- HS has a novel stochastic derivative applied to discrete variables, which uses musician's experiences as a searching direction.
- Certain HS variants do not require algorithm parameters such as HMCR and PAR, thus novice users can easily use the algorithm.

Basic Harmony Search Algorithm

Harmony search tries to find a vector $\mathbf{x}$ that optimizes (minimizes or maximizes) a certain objective function.

The algorithm has the following steps:

Step 1: Generate random vectors ($\mathbf{x}^1 \dots \mathbf{x}^{hms}$) as many as *hms* (harmony memory size), then store them in harmony memory (HM)

$$\mathbf{HM} = \begin{bmatrix} x_1^1 & \cdots & x_n^1 & | & f(\mathbf{x}^1) \\ \vdots & \ddots & \vdots & | & \vdots \\ x_1^{hms} & \cdots & x_n^{hms} & | & f(\mathbf{x}^{hms}) \end{bmatrix}.$$

Step 2: Generate a new vector $\mathbf{x}'$. For each component x_i',

- with probability *hmcr* (harmony memory considering rate), pick the stored value from HM, $x_i' \leftarrow x_i^{int(rand(0,1)*hms)+1}$
- with probability 1 − *hmcr*, pick a random value within the allowed range.

Step 3: Perform additional work if the value in Step 2 came from HM

- with probability *par* (pitch adjusting rate), change x_i' by a small amount: $x_i' \leftarrow x_i' \pm \delta$ for discrete variable; or $x_i' \leftarrow x_i' \pm fw \cdot rand(0,1)$ for continuous variable.
- with probability 1 − *par*, do nothing.

Step 4: If $\mathbf{x}'$ is better than the worst vector $\mathbf{x}^{Worst}$ in HM, replace $\mathbf{x}^{Worst}$ with $\mathbf{x}'$.

Step 5: Repeat from Step 2 to Step 4 until termination criterion (e.g. maximum iterations) is satisfied.

The parameters of the algorithm are

- *hms* = the size of the harmony memory. A typical value ranges from 1 to 100.
- *hmcr* = the rate of choosing a value from the harmony memory. Typical value ranges from 0.7 to 0.99.
- *par* = the rate of choosing a neighboring value. Typical value ranges from 0.1 to 0.5.
- δ = the amount between two neighboring values in discrete candidate set.
- *fw* (fret width, formerly bandwidth) = the amount of maximum change in pitch adjustment.

It is possible to vary the parameter values as the search progresses, which gives an effect similar to simulated annealing.

Parameter-setting-free researches have been also performed. In the researches, algorithm users do not need tedious parameter setting process.

Parallel tempering

Parallel tempering, also known as **replica exchange MCMC sampling**, is a simulation method aimed at improving the dynamic properties of Monte Carlo method simulations of physical systems, and of Markov chain Monte Carlo (MCMC) sampling methods more generally. The replica exchange method was originated by Swendsen , then extended by Geyer and later developed, among others, by Giorgio Parisi. Sugita and Okamoto formulated a molecular dynamics version of parallel tempering.

Typically an MC simulation using a Metropolis-Hastings update consists of a single stochastic process that evaluates the energy of the system and accepts/rejects updates based on the temperature T. At high temperatures updates that change the energy of the system are comparatively more probable. When the system is highly correlated, updates are rejected and the simulation is said to suffer from critical slowing down.

If we were to run two simulations at temperatures separated by a ΔT, we would find that if ΔT is small enough, then the energy histograms obtained by collecting the values of the energies over a set of Monte Carlo steps N will create two distributions that will somewhat overlap. The overlap can be defined by the area of the histograms that falls over the same interval of energy values, normalized by the total number of samples. For $\Delta T = 0$ the overlap should approach 1.

Another way to interpret this overlap is to say that system configurations sampled at temperature T_1 are likely to appear during a simulation at T_2. Because the Markov chain should have no memory of its past, we can create a new update for the system composed of the two systems at T_1 and T_2. At a given Monte Carlo step we can update the global system by swapping the configuration of the two systems, or alternatively trading the two

temperatures. The update can be accepted/rejected with a Metropolis-Hastings criterion, which is

$$p = \min\left(1, \frac{\exp\left(-\frac{E_j}{kT_i} - \frac{E_i}{kT_j}\right)}{\exp\left(-\frac{E_i}{kT_i} - \frac{E_j}{kT_j}\right)}\right) = \min\left(1, e^{(E_i - E_j)\left(\frac{1}{kT_i} - \frac{1}{kT_j}\right)}\right)$$

The detailed balance condition has to be satisfied by ensuring that the reverse update has to be equally likely, all else being equal. This can be ensured by appropriately choosing regular Monte Carlo updates or Parallel Tempering updates with probabilities that are independent of the configurations of the two systems or of the Monte Carlo step.

This update can be generalized to more than two systems.

By a careful choice of temperatures and number of systems one can achieve an improvement in the mixing properties of a set of Monte Carlo simulations that exceeds the extra computational cost of running parallel simulations.

Other considerations to be made: increasing the number of different temperatures can have a detrimental effect, as one can think of the 'lateral' movement of a given system across temperatures as a diffusion process. Set up is important as there must be a practical histogram overlap to achieve a reasonable probability of lateral moves.

The parallel tempering method can be used as a super simulated annealing that does not need restart, since a system at high temperature can feed new local optimizers to a system at low temperature, allowing tunneling between metastable states and improving convergence to a global optimum.