UNIT II SEARCH METHODS AND VISUALIZATION

Search by simulated Annealing – Stochastic, Adaptive search by Evaluation – Evaluation Strategies –Genetic Algorithm – Genetic Programming – Visualization – Classification of Visual Data Analysis Techniques – Data Types – Visualization Techniques – Interaction techniques – Specific Visual data analysis Techniques

---

SEARCH BY SIMULATED ANNEALING

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. For large numbers of local optima, SA can find the global optima. It is often used when the search space is discrete (for example the traveling salesman problem, the boolean satisfiability problem, protein structure prediction, and job-shop scheduling). For problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time, simulated annealing may be preferable to exact algorithms such as gradient descent or branch and bound.

The name of the algorithm comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. Both are attributes of the material that depend on their thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy or Gibbs energy. Simulated annealing can be used for very hard computational optimization problems where exact algorithms fail; even though it usually achieves an approximate solution to the global minimum, it could be enough for many practical problems.

The problems solved by SA are currently formulated by an objective function of many variables, subject to several mathematical constraints. In practice, the constraint can be penalized as part of the objective function.

SIMULATED ANNEALING PROCESS

The state s of some physical systems, 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 simulated annealing heuristic considers some neighboring 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 neighbors of a state

Optimization of a solution involves evaluating the neighbors of a state of the problem, which are new states produced through conservatively altering a given state. For example, in the traveling salesman problem each state is typically defined as a permutation of the cities to be visited, and the neighbors of any state are the set of permutations produced by swapping any two of these cities. The well-defined way in which the states are altered to produce neighboring states is called a "move", and different moves give different sets of neighboring states. These moves usually result in minimal alterations of the last state, in an attempt to progressively improve the solution through iteratively improving its parts (such as the city connections in the traveling salesman problem).

Simple heuristics like hill climbing, which move by finding better neighbor after better neighbor and stop when they have reached a solution which has no neighbors that are better solutions, cannot guarantee to lead to any of the existing better solutions – their outcome may easily be just a local optimum, while the actual best solution would be a global optimum that could be different. Metaheuristics use the neighbors of a solution as a way to explore the solution space, and although they prefer better neighbors, they also accept worse neighbors in order to avoid getting stuck in local optima; they can find the global optimum if run for a long enough amount of time.

Acceptance probabilities

The probability of making the transition from the current state s to a candidate new state s_{new} is specified by an acceptance probability function $P(e, e_{new}, T)$, that depends on the energies $e = E(s)$ and $e_{new} = E(s_{new})$ of the two states, and on a global time-varying parameter T called the temperature. States with a smaller energy are better than those with a greater energy. The probability function P must be positive even when e_{new} is greater than e. This feature prevents the method from becoming stuck at a local minimum that is worse than the global one.

When T tends to zero, the probability $P(e, e_{new}, T)$ must tend to zero if $e_{new} > e$ and to a positive value otherwise. For sufficiently small values of T, the system will then increasingly favor moves that go "downhill" (i.e., to lower energy values), and avoid those that go "uphill." With $T=0$ the procedure reduces to the greedy algorithm, which makes only the downhill transitions.

In the original description of simulated annealing, the probability $P(e, e_{new}, T)$ was equal to 1 when $e_{new} < e$ i.e., the procedure always moved downhill when it found a way to do so, irrespective of the temperature. Many descriptions and implementations of simulated annealing still take this condition as part of the method's definition. However, this condition is not essential for the method to work.

The P function is usually chosen so that the probability of accepting a move decreases when the difference $e_{new} - e$ 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 with regard to 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.

For any given finite problem, the probability that the simulated annealing algorithm terminates with a 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 presents the simulated annealing heuristic as described above. It starts from a state s0 and continues until a maximum of k_{max} steps have been taken. In the process, the call ***neighbour(s)*** should generate a randomly chosen neighbour of a given state ***s***; the call random(0, 1) should pick and return a value in the range [0, 1], uniformly at random. The annealing schedule is defined by the call ***temperature(r)***, which should yield the temperature to use, given the fraction r of the time budget that has been expended so far.

- Let $s = s_0$
- For $k = 0$ through k_{max} (exclusive):
- $T \leftarrow$ temperature$(1 - (k+1)/k_{max})$
- Pick a random neighbour, $s_{new} \leftarrow$ neighbour(s)
- If $P(E(s), E(s_{new}), T) \geq$ random(0, 1):
- $s \leftarrow s_{new}$

- Output: the final state s

Example: Traveling Salesman Problem:

The dataset used in this exercise is a symmetric TSP which means that the distance from city i to city j is the same distance from city j to city i for all cities in the route. The dataset can be found here, along with other TSP datasets.

Each node in the Traveling Salesman problem dataset represents a city. The TSP comes with the following constraints:

- The starting node must be the end node.
- Each node must be visited once and only once.

These nodes are connected to form routes. Each route represents a possible solution/candidate/individual. The fitness of these individuals is taken to be the inverse of the total distance traveled taking that route. The distance traveled between node i and node j is given by:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

where i ≠ j. We take fitness to be the inverse of the distance traveled in the entire route (Fij = 1/dij) because we want to maximize fitness when our algorithm selects a candidate. An example of an individual representation of a route with 10 cities is "1–4–3–2–5–6–7–8–9–10".

Simulated Annealing

This method has been commonly referred to as one of the oldest metaheuristic models[2]. It has great performance in avoiding local minima. In order to do this, the algorithm accepts worse candidates with a probability dependent on the temperature (a control variable) and the fitness difference given by the formula below:

$$p = \begin{cases} 1, & if\ f(y) \geq f(x) \\ e^{-(f(y)-f(x))/T}, & otherwise \end{cases}$$

p — the probability of accepting the new solution candidate, y.

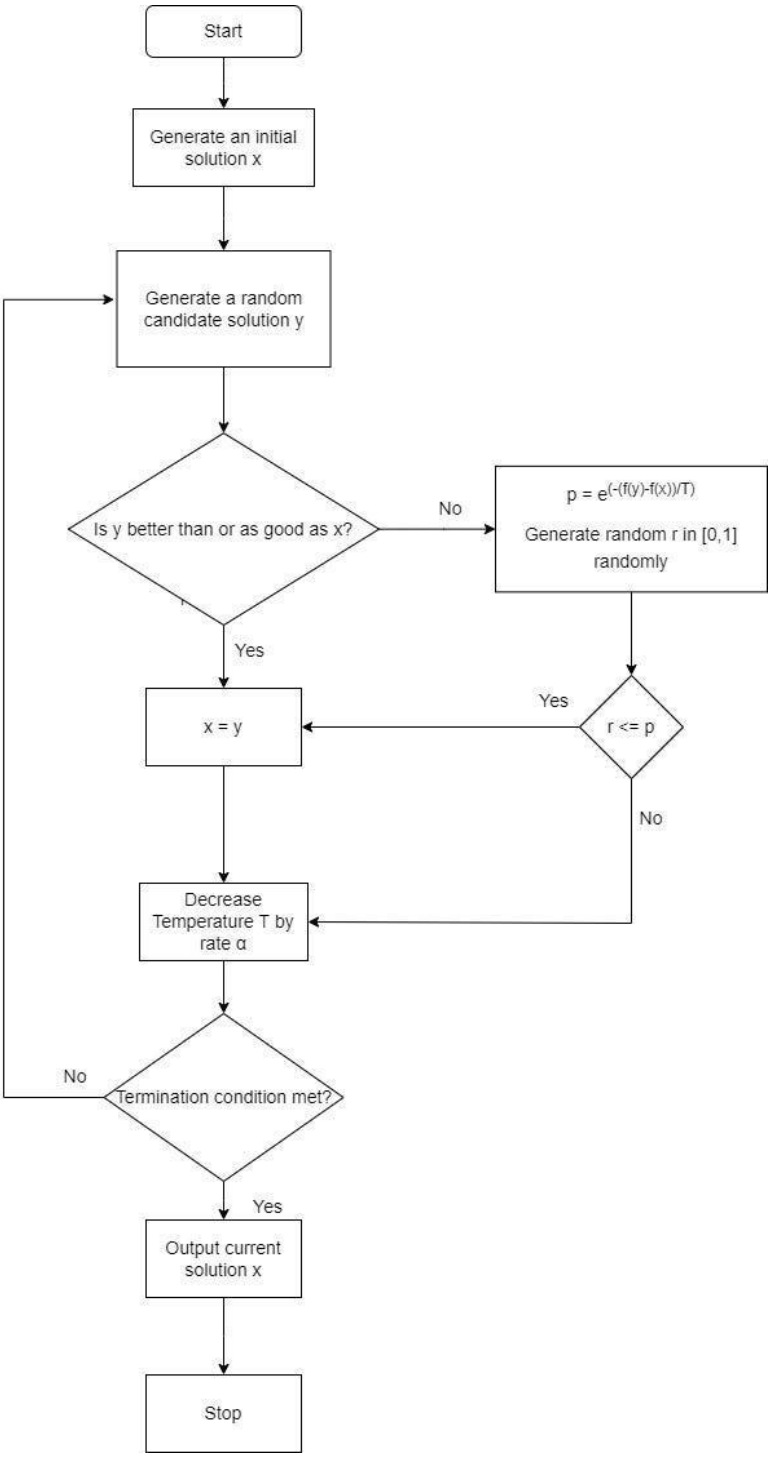
x — initial solution candidate (In this case, that is a route)

y — new solution candidate

f(x) — is the function that measures the performance of the solution candidate (Fitness function)

T — the temperature which is the control parameter.

The Algorithm implemented can be summarized by the flowchart below:



The code below shows how to calculate the fitness:

```
def get_cost(state):
    """Calculates cost/fitness for the solution/route."""
    distance = 0
    for i in range(len(state)):
        from_city = state[i]
        to_city = None
        if i+1 < len(state):
            to_city = state[i+1]
        else:
            to_city = state[0]
        distance += data.get_weight(from_city, to_city)
    fitness = 1/float(distance)
    return fitness
```

Because our probability of accepting worse new candidates is dependent on the temperature, as the temperature cools, eventually we will only accept candidates that are better than the present one.

Generating Candidate Solutions

Because the simulated annealing algorithm compares different candidate solutions and decides which one is taken at each iteration, it is necessary to be able to generate these solutions. In this article, four methods were used, three of which were inspired from[2]:

- **Inverse operator (x):** This changes the order of routes between two randomly selected nodes i and j. Where i,j ≤ 0 ≤ n such that $x^1[i] = x[j]$, $x^1[i+1] = x[j-1]$, etc. Hopefully, an example makes it clearer. Let's say the initial route is '1–2–3–4–5–6', this operator could produce '1–4–3–2–5–6'. The random positions selected in this example are city 2 and city 5. The minimum integer between i and j will be 'i' and the maximum would be taken as 'j'.

```
def inverse(state):
    "Inverses the order of cities in a route between node one and node two"
    node_one = random.choice(state)
    new_list = list(filter(lambda city: city != node_one, state))
    node_two = random.choice(new_list)
```

```
    state[min(node_one,node_two):max(node_one,node_two)]=
    state[min(node_one,node_two):max(node_one,node_two)][::-1]
    return state
```

- **Swap operator (x):** This exchanges the position of two cities in a route. Two positions, i and j are selected at random and the cities in these positions are swapped with each other. '1–2–3–4–5–6' could become '1–5–3–4–2–6'.

```
def swap(state):
    "Swap cities at positions i and j with each other"
    pos_one = random.choice(range(len(state)))
    pos_two = random.choice(range(len(state)))
    state[pos_one], state[pos_two] = state[pos_two], state[pos_one]
    return state
```

- **Insert operator(x):** · This operator selects a city at random position 'i' and moves it to a random position 'j' elsewhere in the route.

```
def insert(state):
    "Insert city at node j before node i"
    node_j = random.choice(state)
    state.remove(node_j)
    node_i = random.choice(state)
    index = state.index(node_i)
    state.insert(index, node_j)
    return state
```

- **Insert subroutes(x):** This is similar to the swap route operator but rather than inserting a city in a different position, a range of cities are selected as a sub route and inserted at a random position.

```
def swap_routes(state):
    "Select a subroute from a to b and insert it at another position in the route"
    subroute_a = random.choice(range(len(state)))
    subroute_b = random.choice(range(len(state)))
    subroute = state[min(subroute_a,subroute_b):max(subroute_a, subroute_b)]
    del state[min(subroute_a,subroute_b):max(subroute_a, subroute_b)]
    insert_pos = random.choice(range(len(state)))
```

```
        for i in subroute:
            state.insert(insert_pos, i)
    return state
```

Termination Condition

While several simulated annealing algorithms terminate once the temperature reaches 0. To have a very large search space, this algorithm only terminates once a candidate has been selected 1500 times and the same fitness score has occurred 150000 times.

STOCHASTIC, ADAPTIVE SEARCH BY EVOLUTION

Computer algorithms modeling the search processes of natural evolution are crude simplifications of biological reality. However, during nearly three decades of research and application, they have turned out to yield robust, flexible and efficient algorithms for solving a wide range of optimization problems.

Variation and Selection:

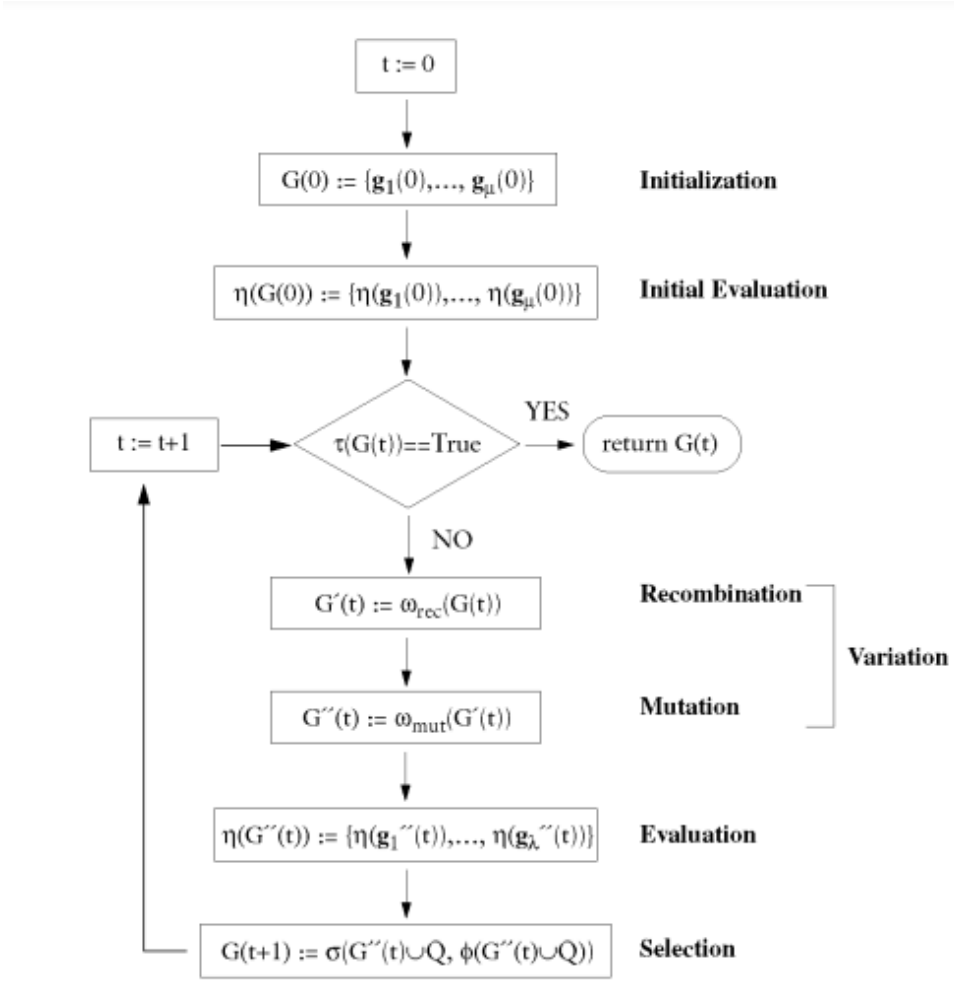
An Evolutionary Algorithm Scheme Evolutionary algorithms (EAs) simulate a collective learning process within a population of individuals. More advanced EAs even rely on competition, cooperation and learning among several populations. Each individual represents a point (structure) in the search space of potential solutions to a specific machine learning problem. After arbitrary initialization of the population, the set of individuals evolves toward better and better regions of the search space by means of partly stochastic processes while the environment provides feedback information (quality, fitness) about the search points:

- **Selection:** It is deterministic in some algorithms, favors those individuals of better fitness to reproduce more often than those of lower fitness.
- **Mutation:** introduces innovation by random variation of the individual structures.
- **Recombination**: which is omitted in some EA realizations, allows the mixing of parental genetic information while passing it to their descendants.

Figure shows the basic scheme of an evolutionary algorithm. Let G denote the search space and let $\eta : G \longrightarrow K$ be the fitness function that assigns a real value $\eta(\sigma_i)$ to each individual structure encoded by a "genome" $g_i \in G$. A population $G(t) = \{g_1\{t),... ,g_\mu(t)\}$ at generation t is described by a (multi) set of μ individuals. From a parent population of size $\mu > 1$ an offspring population of size $\lambda > 1$ is created by means of recombination and mutation at each generation.

A recombination operator $\omega_{rec} : G^{\mu} \rightarrow G^{\lambda}$, controlled by additional parameters $\Theta(\omega_{rec})$, generates λ structures as a result of combining the genetic information of μ individuals.

A mutation operator $\omega_{mut} : G^{\lambda} \rightarrow G^{\lambda}$ modifies a subpopulation of λ individuals, again being controlled by parameters $\Theta(\omega_{mut})$- All the newly created solutions of the population set G" are evaluated. The selection operator σ chooses the parent population for the next generation. The selection pool consists of the λ generated offspring. In some cases, the parents are also part of the selection pool, denoted by the extension set Q = G(t). Otherwise, the extended selection set Q is empty, Q = Ø. Therefore, the selection operator is defined as $\sigma : G^{\lambda} \times K^{\lambda} \rightarrow G^{\mu}$ or $\sigma : G^{\lambda+\mu} \times K^{\lambda+\mu} \rightarrow G^{\mu}$. The termination criterion τ either makes the evolution loop stop after a predefined number of generations, or when an individual exceeding a maximum fitness value has been found.



The Role of Randomness in Evolutionary Learning

The principle dynamic elements that evolutionary algorithms simulate are innovation caused by mutation combined with natural selection. Driving a search process by randomness is, in some way, the most general procedure that can be designed. Increasing order among the population of solutions and causing learning by randomly changing the solution encoding structures may look counterintuitive. However, random changes in combination with fitness-based selection make a formidable and more flexible search paradigm as demonstrated by nature's optimization methods. On average, it is better to explore a search space non-deterministically. This is independent of whether the search space is small enough to allow exhaustive search or whether it is so large that only sampling can reasonably cover it. At each location in a non-deterministic search, the algorithm has the choice of where to go next. Meaning that stochastic search — in combination with a selection procedure which might also be non-deterministic, at least to some degree — is used as a major tool not only to explore but also to exploit the search space. In simulated annealing the Metropolis algorithm also relies on the stochastic search of the Perturb procedure which generates a new solution that competes with the current (best) solution. Evolutionary algorithms use operators like mutation and recombination (in some EA variants even more "genetic" operators are used) to produce new variants of already achieved solutions. These operators rely heavily on randomness: mutation may randomly change parameter settings of a solution encoding vector, and also the mixing of genes by recombination of two solutions is totally non-deterministic. EA-based search algorithms rely on evolution's "creative potential" which is largely due to a controlled degree of randomness.
