IM5110701 Metaheuristics

Dr. Chao-Lung Yang

Department of Industrial Management

National Taiwan University of Science and Technology

Chapter 0: Introduction

Sean Luke, Essentials of Metaheuristics, Second Edition

0.1 What is a Metaheuristic?

- A higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity (Wikipedia)
- Describe a major subfield, indeed the primary subfield, of stochastic optimization (Essentials of Metaheuristics)
  - Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems.

Optimization Problem

- Problem of finding the best solution from all feasible solutions
- Optimization problems can be divided into two categories, depending on whether the variables are continuous or discrete

Continuous Optimization Problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_j(x) = 0, \quad j = 1, \ldots, p \end{aligned}$$

- where
  - $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is the **objective function** to be minimized over the n-variable vector x,
  - $g_i(x) \leq 0$ are called **inequality constraints**
  - $h_j(x) = 0$ are called **equality constraints**, and
  - $m \geq 0$ and $p \geq 0$.

Combinatorial Optimization Problems

- Combinatorial optimization is a subfield of mathematical optimization that consists of finding an optimal object from a finite set of objects, where the set of feasible solutions is discrete or can be reduced to a discrete set.
- Typical combinatorial optimization problems are the travelling salesman problem ("TSP"), the minimum spanning tree problem ("MST"), and the knapsack problem.

Travelling Salesman Problem (1)

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- NP-hard problem in combinatorial optimization
- Integer linear programming formulations
  - Two notable formulations are the Miller–Tucker–Zemlin (MTZ) formulation and the Dantzig–Fulkerson–Johnson (DFJ) formulation.
  - The DFJ formulation is stronger, though the MTZ formulation is still useful in certain settings

Reference: https://en.wikipedia.org/wiki/Travelling_salesman_problem

Travelling Salesman Problem (2)

- Labels the cities with the numbers $1, \ldots, n$ and takes $C_{ij} > 0$ to be the distance from city i to city j

minimize the tour length $\sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}$ $\quad x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$

- Constraints
  - At each vertex is exactly one incoming edge and one outgoing edge

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \ for\ j = 1, \ldots, n \quad \text{and} \quad \sum_{j=1, i \neq j}^{n} x_{ij} = 1 \ for\ i = 1, \ldots, n$$

  - Enough? What do we need to add?

Miller–Tucker–Zemlin Formulation (1)

- for each $i = 2, \ldots, n$ a dummy variable u_i, that keeps track of the order in which the cities are visited, counting from city 1
- the interpretation is that $u_i < u_j$ implies city i is visited before city j.
  - $u_j \geq u_i + 1 \; if \; x_{ij} = 1$
  - $\Rightarrow u_j \geq u_i + x_{ij}?$ (requires $u_j \geq u_i$ when x_{ij} = 0)
- MTZ use the (n-1)(n-2) linear constraints
  - $u_j + (n-2) \geq u_i + (n-1)x_{ij}$ for all distinct $i, j \in \{2, \ldots, n\}$
  - where the constant term $(n-2)$ provides sufficient slack that $x_{ij} = 0$ does not impose a relation between u_i and u_j

C. E. Miller, A. W. Tucker, and R. A. Zemlin. 1960. Integer Programming Formulation of Traveling Salesman Problems. J. ACM 7, 4 (Oct. 1960), 326–329.

Miller–Tucker–Zemlin Formulation (2)

- The way that the u_i variables then enforce that a single tour visits all cities is that they increase by (at least) 1 for each step along a tour, with a decrease only allowed where the tour passes through city 1.
- That constraint would be violated by every tour which does not pass through city 1, so the only way to satisfy it is that the tour passing city 1 also passes through all other cities

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}:$$

$$\begin{aligned}
x_{ij} &\in \{0,1\} && i,j = 1,\ldots,n; \\
u_i &\in \mathbf{Z} && i = 2,\ldots,n; \\
\sum_{i=1, i\neq j}^{n} x_{ij} &= 1 && j = 1,\ldots,n; \\
\sum_{j=1, j\neq i}^{n} x_{ij} &= 1 && i = 1,\ldots,n; \\
u_i - u_j + (n-1)x_{ij} &\leq n-2 && 2 \leq i \neq j \leq n; \\
1 \leq u_i &\leq n-1 && 2 \leq i \leq n.
\end{aligned}$$

Dantzig–Fulkerson–Johnson Formulation

- Labels the cities with the numbers $1, \ldots, n$ and takes $C_{ij} > 0$ to be the distance from city i to city j

minimize the tour length $\sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}$ $\quad x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}:$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \qquad j = 1, \ldots, n;$$

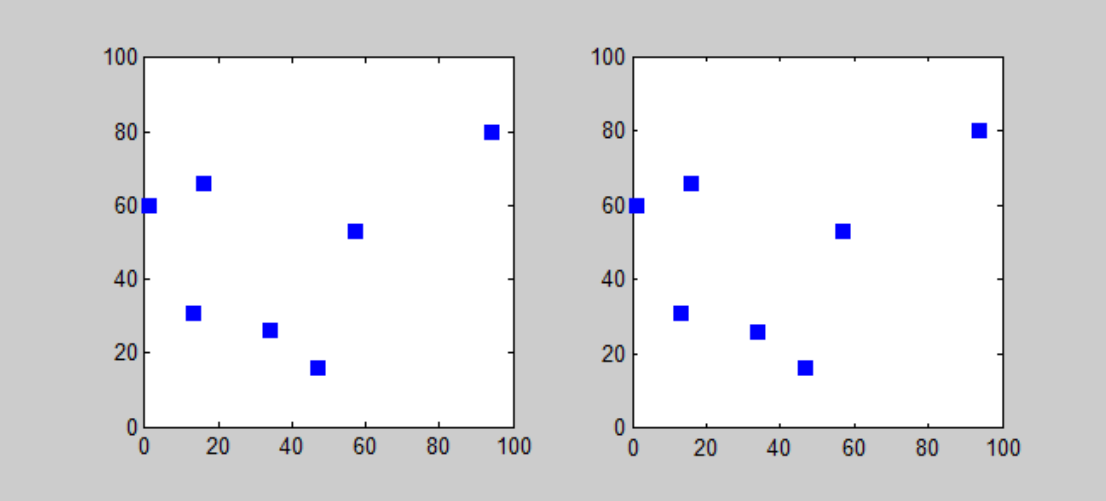
$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n;$$

$$\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1 \qquad \forall Q \subsetneq \{1, \ldots, n\}, |Q| \geq 2$$

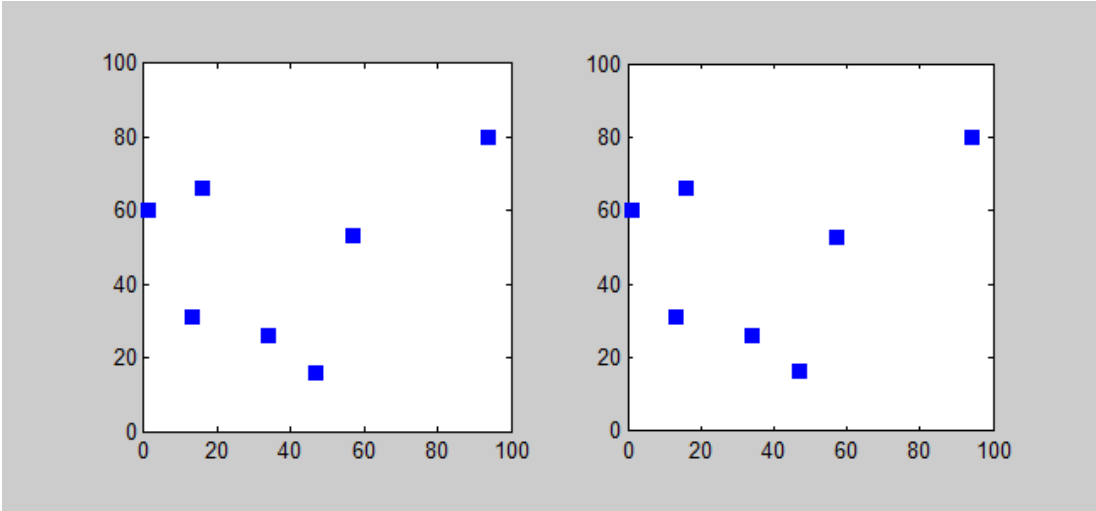
The last constraint of the DFJ formulation ensures no proper subset Q can form a sub-tour, so the solution returned is a single tour and not the union of smaller tours.

Computing a solution

- The running time for this approach lies within a polynomial factor of $O(n!)$
- One of the earliest applications of dynamic programming is the Held–Karp algorithm that solves the problem in time $O(n^2 2^n)$
  - Solution to a symmetric TSP with 7 cities using brute force search. Note: Number of permutations: (7−1)!/2 = 360



  - Solution of a TSP with 7 cities using a simple Branch and bound algorithm. Note: The number of permutations is much less than Brute force search



I know it when I see it problems

- The problems
  - You don't know beforehand what the optimal solution looks like
  - You don't know how to go about finding it in a principled way
  - You have very little heuristic information to go on
  - Brute-force search is out of the question because the space is too large
- But if you're given a candidate solution to your problem, you can test it and assess how good it is.
- That is, you know a good one when you see it.

Random Search vs Hill-Climbing

- Random Search
  - Try random behavior sets as long as you have time, and return the best one you discovered
- Hill-Climbing
  - Start with a random behavior set.
  - Then make a small, random modification to it and try the new version.
  - If the new version is better, throw the old one away. Else throw the new version away.
  - Now make another small, random modification to your current version (which ever one you didn't throw away).
  - If this newest version is better, throw away your current version, else throw away the newest version.
  - Repeat as long as you can.

Hill-Climbing (Cont'd)

- Hill-climbing is a simple metaheuristic algorithm. It exploits a heuristic belief about your space of candidate solutions which is usually true for many problems:
  - Similar solutions tend to behave similarly (and tend to have similar quality), so small modifications will generally result in small, well-behaved changes in quality, allowing us to "climb the hill" of quality up to good solutions.
- This heuristic belief is one of the **central defining features of metaheuristics**:
  - Nearly all metaheuristics are essentially elaborate combinations of hill-climbing and random search.

Inverse Problems

- "I know it when I see it" problems tackled by metaheuristics are a subclass of inverse problems.
- An inverse problem is one in which you have a test function f which takes a candidate solution and produces an assessment of it, but in which it's difficult or impossible to construct the inverse function f^{-1} which takes an assessment and returns a candidate solution which would have had that assessment.

So, Metaheuristics is …

- Optimization methods (such as metaheuristics) are designed to overcome inverse problems.
- Many classic optimization techniques, such as **Gradient Ascent** (Algorithm 1) make strong assumptions about the nature of f :
  - for example, that we also know its first derivative f'.
- Metaheuristics make far weaker assumptions, and sometimes make none at all.
  - This means that metaheuristics are very general, but also means that they're often best thought of as last-ditch methods, used when *no other known technique works*.

0.2 Algorithms

- **Parameters:** appear first followed by a blank line
- **Procedure:** static global variables are defined which appear at the beginning and are labelled **global**
  - Perform once per generation or perform every n iterations, versus perform each time.

Bubble Sort

Algorithm 0 *Bubble Sort*

```
1: v⃗ ← ⟨v_1, ..., v_l⟩ vector to sort        ▷ User-provided parameters to the algorithm appear here
                                                ▷ Then a blank space
2: repeat                                       ▷ Algorithm begins here
3:     swapped ← false                          ▷ ← always means "is set to"
4:     for i from 1 to l − 1 do                 ▷ Note that l is defined by v_l in Line 1
5:         if v_i > v_{i+1} then
6:             Swap v_i and v_{i+1}
7:             swapped ← true
8: until swapped = false                        ▷ = means "is equal to"
9: return v⃗                                    ▷ Some algorithms return nothing, so there is no return statement
```

0.3 Notation (1)

- **Numbers** and **booleans** are denoted with lower-case letters, greek symbols, or words $(n, \lambda, \min, popsize)$
  - The symbol $\longleftarrow$ always means "is set to", and the symbol $=$ usually means "equals".
- **Candidate solutions** (sometimes called **individuals, particles,** or **trails**) are indicated with upper-case letters or words $(Best, S, P_i)$
  - Others consist of a number of **components**, often designated $C_1, \ldots, C_j$.
  - Candidate solutions may be associated with some kind of **quality** (**fitness**), usually via a function like Quality(S) or Fitness(P_i)
  - Quality can be set as well. Usually quality is a single number; but can in some cases (for multiobjective optimization) be a group of numbers called **objectives**.
  - The value of objective O_j, assigned to individual P_i, is accessed via a function like ObjectiveValue(O_j,P_i).

0.3 Notation (2)

- **Collections** (or bags, or groups, or pools, or lists, or multisets) are groups of objects where the objects usually don't have to be unique.
  - Read-only
  - $\{P_1, \ldots, P_n\}$
  - (**for** each element $P_i \in P$ **do** … P_i) or this (**for** i from 1 to n **do** … P_i)
  - $P \longleftarrow P \cup Q, P \longleftarrow P \cup \{R_j\}, P \longleftarrow P - \{P_2\}$
  - Populations
    - $P^{(1)}, \ldots, P^{(n)}$
    - An individual numbered j in population $P^{(i)}$ would be $P_j^{(i)}$

0.3 Notation (3)

- Vectors **are modifiable**
  - $\langle x_1, \ldots, x_n \rangle$
  - An element in a vector can be replaced with another object at the same location
- Tuples are vectors with named slots rather than numbered slots.
  - $\vec{t} \longleftarrow \langle t_{lock}, t_{data} \rangle$
- **Arrays** or **Matrices**
  - $A_{i,j}$

Chapter 1: Gradient-based Optimization

Sean Luke, Essentials of Metaheuristics, Second Edition

Gradient Ascent: Finding the Maximum of a Function

- To Maximum $f(x)$
  - You don't need to know $f(x)$, but it assumes you can compute slope of x, $f'(x)$
- Steps
  - Start with an arbitrary value of x
  - $x \longleftarrow x + \alpha f'(x)$ where α is a very small positive value
    - If $f'(x)$ is positive, x will increate
- Ultimately x will move up the function until it is at the peak, at which point the slope is zero and x won't change any more.

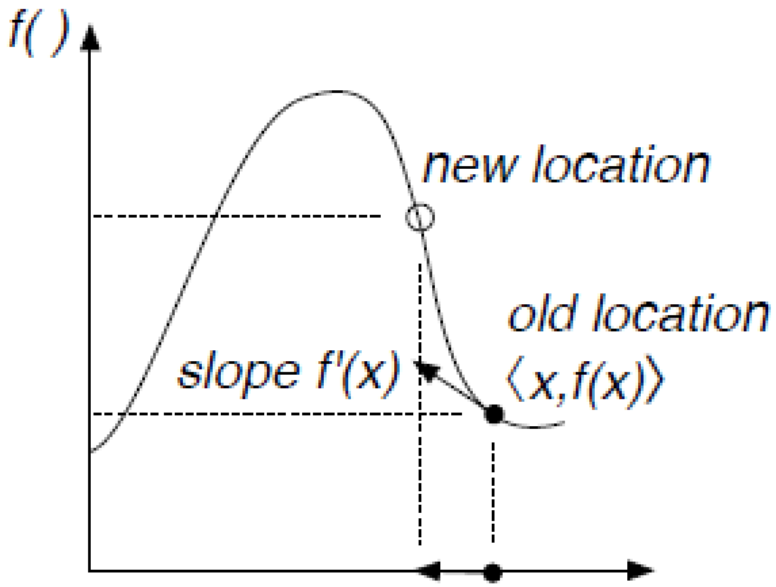


Figure 1 Gradient Ascent with a negative slope. x is decreasing.

Gradient Ascent: Maximum of a Multidimensional Function

- Replace x with the vector $\vec{x}$,
- Replace slope $f'(x)$ with the gradient of $\vec{x}$, $\nabla f'(x)$
  - The gradient is simply a vector where each element is the slope of $\vec{x}$ in that dimension, that is, $\left\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right\rangle$

Algorithm 1 *Gradient Ascent*

```
x⃗ ← random initial vector
repeat
    x⃗ ← x⃗ + α∇f(x⃗)                    ▷ In one dimension: x ← x + αf′(x)
until x⃗ is the ideal solution or we have run out of time
return x⃗
```

Gradient Ascent: When to Stop?

- "the ideal solution" or "we have run out of time".
- How do we know that we've got the ideal solution?
  - Typically when the slope is 0.
  - How about saddle point?
- Convergence time?
- Overshoot!

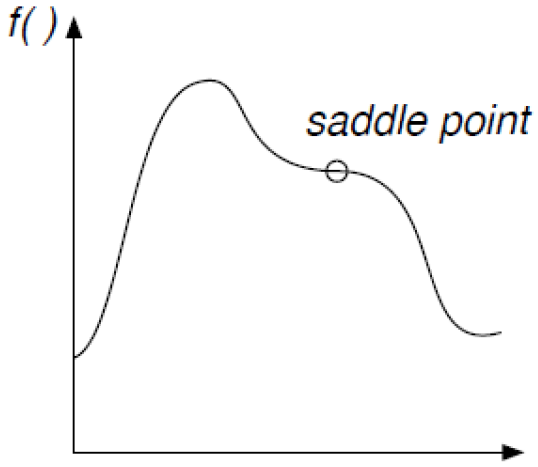


Figure 2 A saddle point.

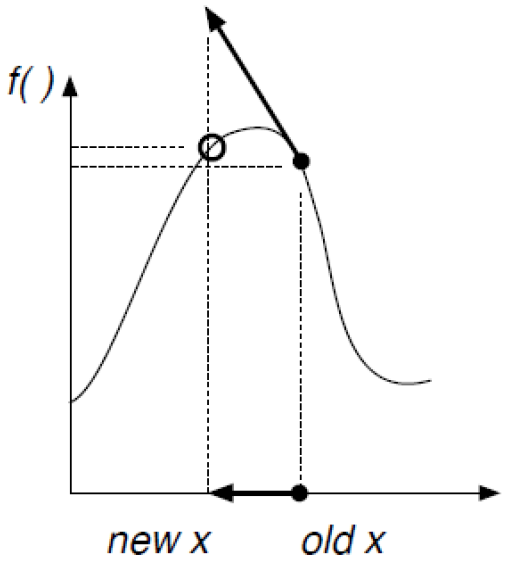


Figure 3 Gradient Ascent overshooting the maximum.

Gradient Ascent: Deal with Overshoot

- The size of the jumps Gradient Ascent makes is entirely based on the current slope
- Adjusting the value of α
  - Very small value of α and Gradient Ascent won't overshoot hills but it may take a long time to march up the hills and converge to the top.
  - But a very big value ofα will cause Gradient Ascent to constantly overshoot the hills which also causes it to take a long time to converge to the maximum, if at all.
  - We're looking for a value of a which is "just right". HOW?

Newton's Method: A Variation on Gradient Ascent

- If we could compute not only $f'(x)$ but also $f''(x)$, we could use Newton's Method
- This variation on Gradient Ascent includes an additional $\frac{-1}{f''(x)}$
  - $x \longleftarrow x - \alpha \frac{f'(x)}{f''(x)}$
  - Dampens α as we approach a zero slope
  - Head towards any kind of zero-slope point (maxima, minima, or saddle points)

Newton's Method : Maximum of a Multidimensional Function

- First derivative $f'(x) \longrightarrow \nabla f'(\vec{x})$
- Second derivative $f''(x) \longrightarrow$ **Hessian** $\boldsymbol{H_f}(\vec{x})$
  - $\boldsymbol{H_f}(\vec{x})$ is consist of partial second derivatives along each dimension

$$H_f(\vec{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1}\frac{\partial f}{\partial x_1} & \frac{\partial}{\partial x_1}\frac{\partial f}{\partial x_2} & \cdots & \frac{\partial}{\partial x_1}\frac{\partial f}{\partial x_n} \\ \frac{\partial}{\partial x_2}\frac{\partial f}{\partial x_1} & \frac{\partial}{\partial x_2}\frac{\partial f}{\partial x_2} & \cdots & \frac{\partial}{\partial x_2}\frac{\partial f}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_n}\frac{\partial f}{\partial x_1} & \frac{\partial}{\partial x_n}\frac{\partial f}{\partial x_2} & \cdots & \frac{\partial}{\partial x_n}\frac{\partial f}{\partial x_n} \end{bmatrix}$$

Algorithm 2 *Newton's Method (Adapted for Optima Finding)*

1: $\vec{x} \leftarrow$ random initial vector
2: **repeat**
3: $\quad \vec{x} \leftarrow \vec{x} - \alpha \boxed{[H_f(\vec{x})]^{-1}} \nabla f(\vec{x})$ ▷ In one dimension: $x \leftarrow x - \alpha \frac{f'(x)}{f''(x)}$
4: **until** $\vec{x}$ is the ideal solution or we have run out of time
5: **return** $\vec{x}$

Inverse of $\boldsymbol{H_f}(\vec{x})$

Newton's Method (cont'd)

- Because Newton's method employs the second derivative, it generally converges faster than regular Gradient Ascent
- Newton's method also gives us the information to determine if we're at the top of a local maximum (as opposed to a minimum or saddle point) because at a maximum, $f'(x)$ is zero and $f''(x)$ is negative.
- BUT….still get caught in local optima

Local Optima vs Global Optima

- Local optima of a function are the optima (in our case, maxima) of a local region.
- Global optima are the optima of the entire function.
- Gradient Ascent and Newton's Method are local optimization algorithms

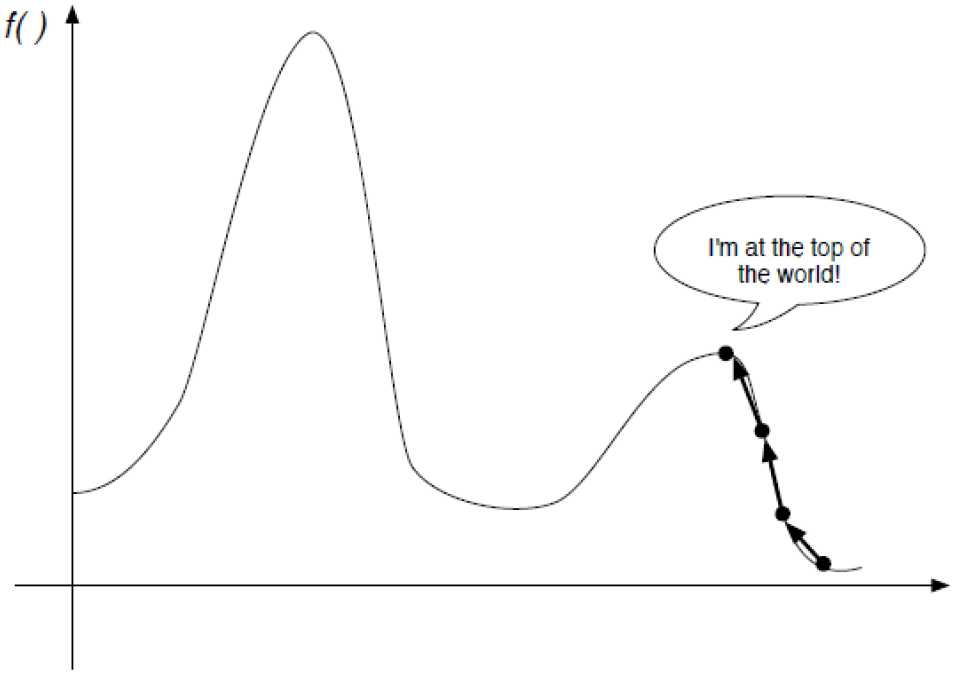


Figure 5 Gradient Ascent stuck in a local optimum.

How to escape local optima?

- Change α to a sufficiently large value that the algorithm potentially overshoots not only the top of its hill but actually lands on the next hill.
- Alternatively, we could put Gradient Ascent in a big loop: each time we start with a random starting point, and end when we've reached a local optimum
  - We keep trying over and over again, and eventually return the best solution discovered (compute $f(x)$)

Global Optimization Algorithm

Algorithm 3 *Gradient Ascent with Restarts*

```
x⃗ ← random initial value
x⃗* ← x⃗                                   ▷ x⃗* will hold our best discovery so far
repeat
    repeat
        x⃗ ← x⃗ + α∇f(x⃗)                    ▷ In one dimension: x ← x + αf′(x)
    until ||∇f(x⃗)|| = 0                    ▷ In one dimension: until f′(x) = 0
    if f(x⃗) > f(x⃗*) then                   ▷ Found a new best result!
        x⃗* ← x⃗
    x⃗ ← random value
until we have run out of time
return x⃗*
```

$$f(x_1, x_2) = 0.25x_1^4 - 0.5x_1^2 + 0.1x_1 + 0.5x_2^2 \tag{13}$$

Bound at $[-10 \leq x_i \leq 10]$. Global minimum solution at $x^* = (-1.0465, 0)$, where $f(x^*) = -0.3523$.

Notes of Gradient Ascent with Restarts

- A global optimization algorithm is guaranteed to find the global optimum if it runs long enough
- And at the limit, at some point Gradient Ascent with Restarts will discover the optimum because, at the very least, some restart will randomly land right on the optimum, just like random search.
- More realistically, a random restart will eventually start on the globally optimal hill, allowing gradient ascent to climb to the optimum.
- Note that we'll likely never wind up with $f'(x)$ precisely equal to 0.
  - So we'll have to fudge it: if $-\epsilon < f'(x) < \epsilon$ for some very small value of ϵ, we'll consider that "close enough to zero"

Code Evaluation

- Please use the computer language you are familiar with to implement "Gradient Ascent with Restarts"
  - You can start with one-dimension problem
  - Think about how to handle Multidimensional Function?
- If you use Python, here is resource (not limited to these) :
  - https://realpython.com/gradient-descent-algorithm-python/
  - https://www.geeksforgeeks.org/how-to-implement-a-gradient-descent-in-python-to-find-a-local-minimum/