Modelling Transportation Systems

Heuristic Algorithm

Dr Zhiyuan Liu
Email: zhiyuanl@seu.edu.cn

Solution Algorithm

- **Learning Objectives**

- Know the concept of enumeration, heuristics and exact algorithm.
- Understand the connection and difference between P, NP, NP-C and NP-hard.
- Master the application and coding of GA and ABC algorithms.

A Good Solution

Research Viewpoint

Industrial Viewpoint

- A solution that is optimal
- A solution that is near a provable optimality bound
- A solution that is better than the solutions of other studies
- A solution that is better than basic heuristics
- A solution that is better than the status quo
- A solution that is reasonable for managers and there is no obvious improvement of the solution

Reasonable Computation Time

- The length of reasonable time is problem-dependent.
  - If we aim to design a bus route (planning level) that will be operated for one year, then 2 days could be a reasonable time.
  - If we aim to design the route for delivery trucks of a furniture company on a daily basis, then 2 hours (computed during e.g. 1 a.m. to 3 a.m.) is a reasonable time.
  - If we aim to determine the routing for KFC delivery, then 5 seconds is a reasonable time.

Reasonable Computation Time

- In practice, even if for the bus route design problem, a shorter computational time is still highly desirable, because Mathematical Model ≠ Reality, and managers would like to see the results of many different cases with different parameters.
- The cost associated with computational time may also be incorporated. However, today because of the low price of PC, this cost is generally negligible (except for large projects such as weather forecasting).

P, NP, NP-C and NP-Hard

P=NP?

- 千禧年大奖难题(Millennium Prize Problems), 又称世界七大数学难题， 是七个由美国克雷数学研究所(Clay Mathematics Institute,CMI) 于2000年5月24日公布的7个数学猜想。拟定这7个问题的数学家之一是怀尔斯，费马大定理这个有300多年历史的难题没被选入的唯一理由就是已经被他解决了。
- 根据克雷数学研究所订定的规则，任何一个猜想的解答，只要发表在数学期刊上，并经过两年的验证期，解决者就会被颁发一百万美元奖金。这些难题是呼应1900年德国数学家大卫·希尔伯特在巴黎提出的23个数学问题。
- P是否等于NP的问题，即能用多项式时间验证解的问题是否能在多项式时间内找出解，是计算机与算法方面的重大问题，是斯蒂文·考克（StephenCook）于1971年陈述的。

Problems, Algorithms and Complexity

❑What is the problem （问题）?

- In definition, a problem will be a general question to be answered, usually possessing several parameters, or free variables, whose values are left unspecified.
- A problem is described by giving:
- (1) a general description of all its parameters, and
- (2) a statement of what properties the answer, or solution, is required to satisfy.
- An ***instance*** （实例）of a problem is obtained by specifying particular values for all the problem parameters.

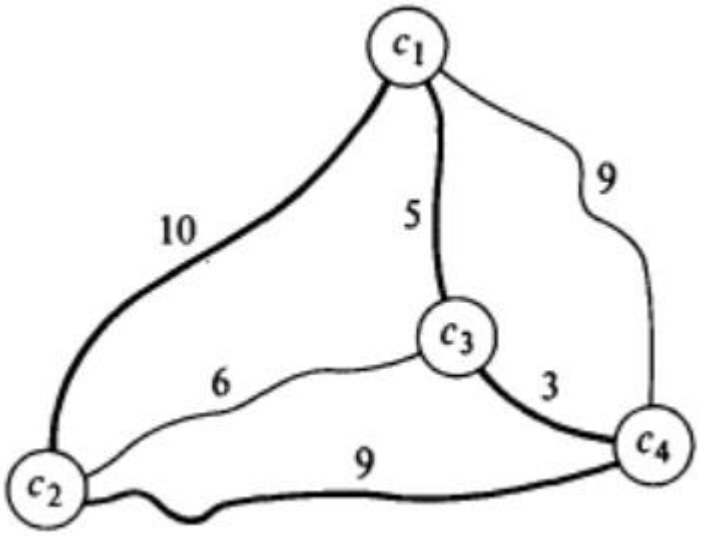
Problems, Algorithms and Complexity

Traveler Salesman Problem (TSP)

- Given a list of cities and their pairwise distances, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city.
  - The distance can be symmetric or asymmetric.
  - Triangular law is usually assumed. (满足三角不等式：两边之和大于第三边)

An instance

- A particular trip: 2—1—4—3—2



Problems, Algorithms and Complexity

- **What is the Algorithm?**
- Algorithms are general, step-by-step procedures for solving problems.
- Almost all the common problems can be solved by an algorithm. Then, how to gauge the efficiency of an algorithm?
  - Convergent Speed
  - Space Complexity v.s. Time Complexity

Problems, Algorithms and Complexity

- Convergent speed
  - Sublinear, linear, super-linear
  - Method of Successive Average, Bi-section method, Newton's method

Suppose that the sequence $\{x_k\}$ converges to the number L.

We say that this sequence **converges linearly** to L, if there exists a number $\mu \in (0, 1)$ such that

$$\lim_{k\to\infty} \frac{|x_{k+1} - L|}{|x_k - L|} = \mu.$$

The number μ is called the *rate of convergence*.

If the sequence converges, and

- $\mu = \mu_k$ varies from step to step with $\mu_k \to 0$ for $k \to \infty$, then the sequence is said to **converge superlinearly**.
- $\mu = \mu_k$ varies from step to step with $\mu_k \to 1$ for $k \to \infty$, then the sequence is said to **converge sublinearly**.

Problems, Algorithms and Complexity

❑Space Complexity

- The ***Space Complexity*** is used to measure the amount of space, or memory required by an algorithm to solve a given problem.
- The Space Complexity of an algorithm is expressed using big O notation (short for Order, 量级), $S(n)=O(f(n))$, where n is the scale of the problem and $f(n)$ is the function of the amount of space with respect to n.

Problems, Algorithms and Complexity

Time complexity

- In computer science, the time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input.
- The time complexity of an algorithm is also expressed using big *O* notation , which excludes coefficients and lower order terms. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$ for any n, the asymptotic time complexity is $O(n^3)$.

$$O(1) < O(\log_n) < O(n) < O(n\log_n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

Time complexity

- **How to calculate the time complexity?**

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform.

Since an algorithm's performance time may vary with different inputs of the same size, one commonly uses the worst-case time complexity of an algorithm.

- 时间复杂度是总运算次数表达式中受n的变化影响最大的那一项(不含系数)。

Time complexity

- Example 1

```
For (i=1; i<=n; i++)
    s++;
end
```

Numbers of iterations: n
Time complexity: $O(n)$.

- Example 2

```
For (i=1; i<=n; i++)
    For (j=1; j<=n; j++)
        s++;
    end
end
```

Numbers of iterations: $n*n$,
Time complexity: $O(n^2)$.

Time complexity

- Example 3

```
For (i=1; i<=n; i++)
    For (j=i; j<=n; j++)
        s++;
    end
end
```

Number of iterations: $n+n-1+\ldots+1 \approx n^2/2$.
Time complexity: $O(n^2)$.
*计算时间复杂度不考虑系数

- Example 4

```
For (i=1; i<=n; i++)
    For (j=1; j<=n; j++)
        For (k=1; k<=n; k++)
            s++;
…
```

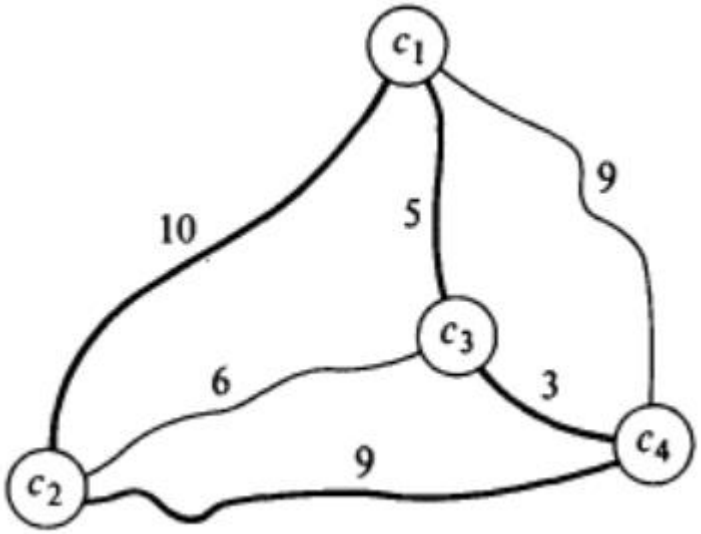
Number of iterations:
$1^2+2^2+3^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3$.
Time complexity: $O(n^3)$.

Time complexity

- 时间复杂度并不是表示一个程序解决问题需要花多少时间，而是当问题规模扩大后，程序需要的时间长度增长得有多快。如果不管数据有多大，程序处理花的时间始终是那么多的，我们就说这个程序很好，具有O(1)的时间复杂度，也称常数级复杂度；如果数据规模变得有多大，花的时间也跟着变得有多长，这个程序的时间复杂度就是O(n)；如果数据扩大2倍，时间变慢4倍的，属于$O(n^2)$的复杂度。
- 不会存在$O(2*n^2)$的复杂度，因为前面的那个“2”是系数，根本不会影响到整个程序的时间增长。因此，我们会说，一个$O(0.01*n^3)$的程序的效率比$O(100*n^2)$的效率低，尽管在n很小的时候，前者优于后者，但后者时间随数据规模增长得慢，最终$O(n^3)$的复杂度将远远超过$O(n^2)$。同样，$O(n^{100})$的复杂度小于$O(1.01^n)$的复杂度。

TSP-Enumeration Method

- The current algorithms to solve TSP do not have polynomial time complexity.
- Enumerate all the possible tours is a **Permutation Problem** . Thus, this example network contains $A_3^3 = 3! = 6$ tours.



$10! = 3628800$

$20! = 2432902008176640000$

$100! = 9.3326215444 \times 10^{157}$

$1000! = 4.0238726008 \times 10^{2,567}$

Time complexity: $O(n!)$

- This approach quickly becomes untrackable when n increases.

TSP-Dynamic Programming Method

Characterization of the optimal solution

Given $S \subseteq V$ with $1 \in S$ and given $j \neq 1, j \in S$, let $C(S, j)$ be the shortest path that starting at 1, visits all nodes in S and ends at j.

Notice:

- If $|S| = 2$, then $C(S, k) = d_{1,k}$ for $k = 2, 3, \ldots, n$
- If $|S| > 2$, then $C(S, k) =$ the optimal tour from 1 to m, $+d_{m,k}$,

  $\exists m \in S - \{k\}$

Recursive definition of the optimal solution

$$C(S,k) = \begin{cases} d_{1,m} & \text{if } S = \{1,k\} \\ \min_{m \neq k, m \in S}[C(S - \{k\}, m) + d(m,k)] & \text{otherwise} \end{cases}$$

TSP-Dynamic Programming Method

The optimal solution

function algorithm $TSP(G, n)$
  for $k := 2$ **to** n **do**
    $C(\{i, k\}, k) := d_{1,k}$
  end for
  for $s = 3$ **to** n **do**
    for all $S \subseteq \{1, 2, \ldots, n\} || S || = s$ **do**
      for all $k \in S$ **do**
        $\{C(S, k) = \min_{m \neq k, m \in S}[C(S - \{k\}, m) + d_{m,k}]\}$
        $opt := \min_{k \neq 1}[C(\{1, 2, 3, \ldots, n\}, k) + d_{1,k}$
      end for
    end for
  end for;
  return (opt)
end

Complexity:

Time: $(n-1)\sum_{k=1}^{n-3}\binom{n-2}{k} + 2(n-1) \sim O(n^2 2^n) << O(n!)$

Space: $\sum_{k=1}^{n-1} k\binom{n-1}{k} = (n-1)2^{n-2} \sim O(n2^n)$

P & NP

- **Polynomial time** $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$

An algorithm is said to be of polynomial time if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm, i.e., $T(n) = O(n^k)$ for some constant k.

- Some examples of polynomial time algorithms:

- The *sorting algorithm* on n integers performs An^2 operations for some constant A. Thus it runs in time $O(n^2)$ and is a polynomial time algorithm.
- All the *basic arithmetic operations* (addition, subtraction, multiplication, division, and comparison) can be done in polynomial time.
- *Maximum matchings* in graphs can be found in polynomial time.

P & NP

- ***P*** is the set of decision problems that can be solved *in polynomial time* by a deterministic Turing machine.
- ***P*** is the set of problems that can be solved quickly.
  - LP and the shortest path problem are in ***P***.
  - Example: Given n numbers, is the largest one larger than 100?
  - We could obtain the largest one after n-1 comparisons: $O(n)$
- Literally, ***NP*** (Non-Deterministic Polynomial) is set of problems solvable *in polynomial time* by a non-deterministic Turing machine.
- 即幸运的情况下，可以在多项式的时间里猜出一个解的问题

P & NP

- ***NP (Non-deterministic polynomial)*** is the set of problems such that given a solution, we can check whether the solution is feasible and calculate its objective function value in polynomial time with regards to the size of the problem.
- Example of checking a solution: In the TSP, to check **a particular tour is shorter than 100**, then I could verify it by
  - n-1 additions. $O(n)$
  - One comparison.
- All problems in ***P*** are also in ***NP***. TSP is in ***NP***.

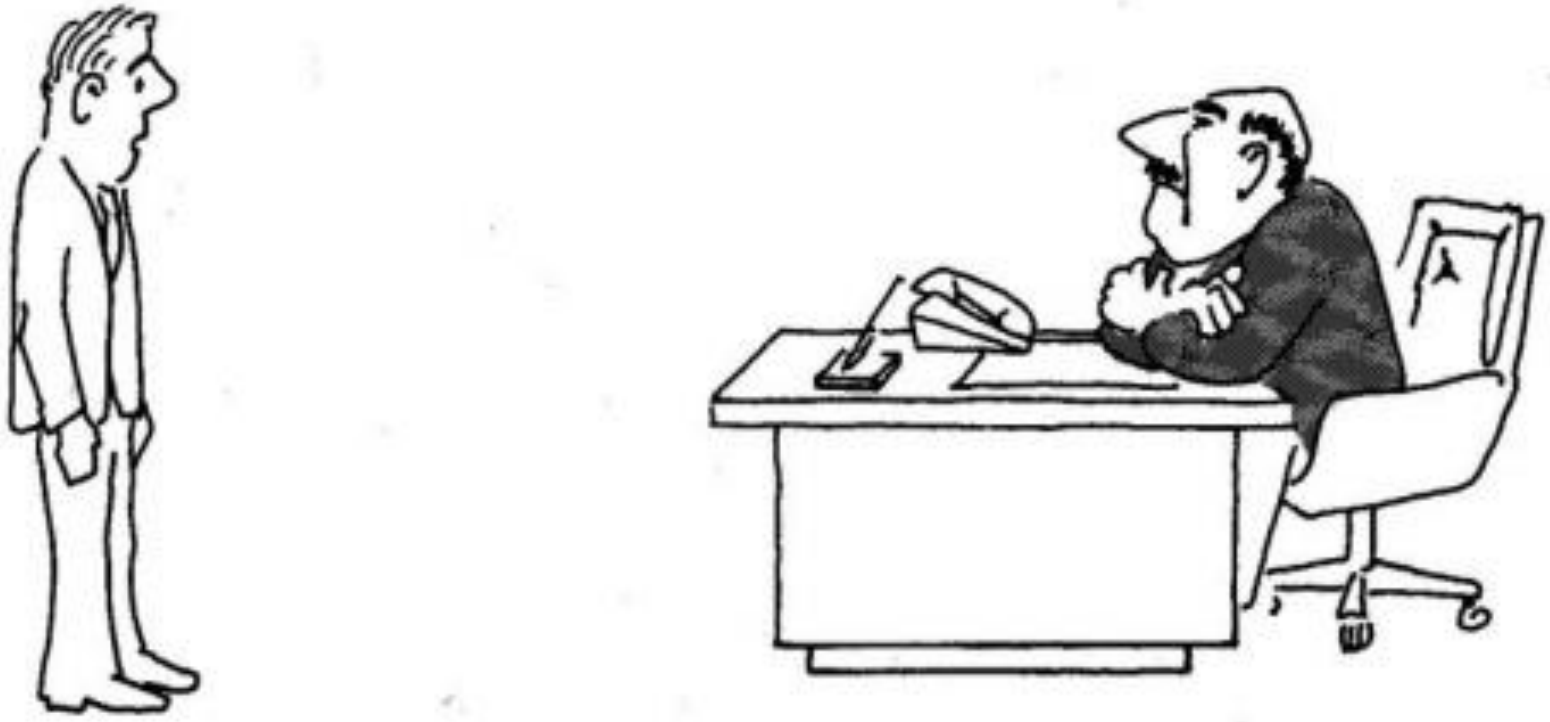
P & NP

- The decision problem and the optimization problem are much more difficult than verifying a given solution.
- TSP:
  - Decision problem: Is there a tour shorter than 100?
  - Optimization problem: Find the shortest tour.
- So, we don't know whether we can find the optimal solution and make a decision efficiently (in polynomial time complexity). Can any NP problem be solved in P time?
- So, is NP the complementary set of P? We don't know!

P & NP

- There are some problems that cannot even be checked in polynomial time, which is more complicated than the NP problems. We don't discuss about these problems, because it seems impossible to find efficient algorithms (polynomial time complexity) for these problems.
- Namely, it's pointless to discuss whether these problems = P?
- So, the current key topic is: NP=P?
- 证明或推翻NP=P!

An interesting story: three answers



"I can't find an efficient algorithm, I guess I'm just too dumb."

An interesting story: three answers

"I can't find an efficient algorithm, because no such algorithm is possible!"

An interesting story: three answers

"I can't find an efficient algorithm, but neither can all these famous people."

NP-hard and NP-C

变化法则：
对任意一个程序A的输入，都能变换成问题B的输入，而且得到A的解

Reduction (约化、归约)

- **Definition:** In computational complexity theory, a ***reduction*** is an algorithm for **transforming** one problem into another problem. A reduction from one problem to another may be used to show that the second problem is at least as difficult as the first.
- “A can be reduced to B” equals “B is at least as difficult as A”.
- 一个问题A可以约化为问题B的含义即是，可以用问题B的解法解决问题A，或者说，问题A可以“变成”问题B。
- “问题A可约化为问题B”有一个重要的直观意义：B的时间复杂度高于或者等于A的时间复杂度。也就是说，问题A不比问题B难。
- **约化具有传递性。**如果问题A可约化为问题B，问题B可约化为问题C，则问题A一定可约化为问题C。

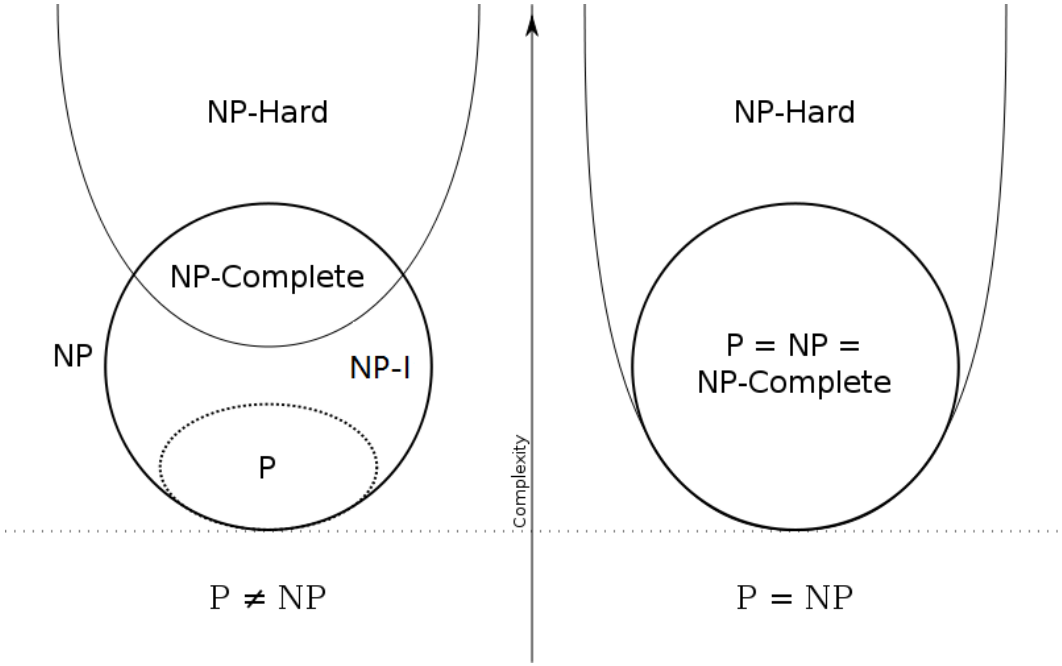
NP-hard and NP-C

- A problem is **NP-hard** if a polynomial-time algorithm for it would imply a polynomial time algorithm for every problem in **NP**. (all the NP problems can be reduced to it).
  - Literally, a problem is Hard, comparing with the NP problems.
- A problem is **NP-C** if it is both **NP-hard** and an element of **NP**. In other words, the **NP-C** is a set of problem of the "hardest" **NP** problems.
- We can also say that: *all the problems in **NP** can be **reduced** to **NP-C**.*

Definitions of P, NP, NP-C and NP-hard

- **Set A**: the set of problems that can be **solved** in polynomial time. (e.g. $1+2+\ldots+n$)
- **Set B**: is the set of problems such that given any solution, we can **check** whether the solution is feasible and calculate its objective function value in polynomial time with regard to the size of the problem. (e.g. find the largest prime number smaller than a given number, n)
- **Set C**: if "a polynomial-time algorithm exists for it" would imply that a polynomial time algorithm exists for every problem in NP.
- P: $A=A\cap B$; NP: B
- NP-hard: C; NP-Complete (NP-C): $B\cap C$

Venn Diagram of P and NPs



Is P=NP?

- Is P=NP?
  - Could all the NP problems be solved in polynomial time?
- NP-I (Intermediate)

If P≠NP, then NP-I exists. NP-I is NP, but not P and NP-C

It ***can be checked*** in polynomial time but ***cannot be solved***, and not all NP can be reduced to it.

NP-hard and NP-complete

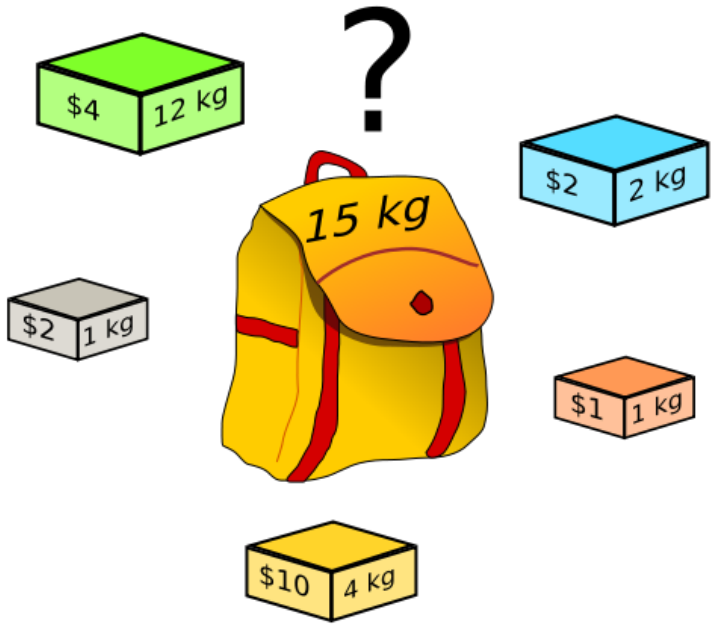
- There are many problems that have been proved to be NP-hard, or NP-Complete, or NP, or P.
- Example:
  - **NP-C:** TSP, Graph Coloring Problem, Scheduling Problem, Knapsack Problem, Hamilton Problem, Node Coverage Problem
  - **P:** The shortest path problem, LP models with a polynomial number of variables and constraints.
- There are still many problems whose categories are unknown.

NP-hard

- To prove that a problem is NP-hard, we usually **reduce** a known NP-C or NP-hard problem to it.
- **Example 1:**
  - We know that TSP is NP-C, prove that the problem of how many tours are shorter than 100 is NP-hard.
  - Proof: it is easy to see that if the latter problem could be solved in polynomial time, then TSP can also be solved in polynomial time. Hence, the latter problem is NP-hard.

NP-hard

- The knapsack problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- The knapsack problem is NP-C



NP-hard

- **Example 2:**
  - Prove that integer programming problems are NP-hard.
  - Proof: The knapsack problem can be formulated as an integer programming problem with a polynomial number of variables and constraints.
- When we are mentioning the hardness of a problem class, we refer to the most difficult instance of the problem class.
  - One dimensional TSP is very simple and is in P.
  - Knapsack problem is also very simple if the higher the value of an item is, the smaller its weight is.

NP-hard and NP-C

- **How to prove a problem is NP-C?**

From the definition of NP-C, we can find that if a problem is NP-C, it should satisfy two conditions:

- This problem is NP;
- All the problems in NP can be reduced to it.

Therefore, if we want to prove a problem is NP-C, we should:

- **Step 1:** Prove that it is NP;
- **Step 2:** Prove that a known NP-C problem can be reduced to it. (This problem is at least as hard as the known NP-C problem).

What if we have a problem that is NP-hard?

- We should stop attempting to find a polynomial algorithm for it, unless we aim to address the fundamental problem of whether P=NP.
- We could still design good algorithms that efficiently obtain the optimal solution for many instances of real-sized problems (for example, commercial mixed-integer programming solvers).
- Otherwise, we must turn to heuristic methods.

Example for Proof of NP-Hardness: Rural Bus Route Design Problem

- **Introduction (reading materials)**

In this study, the authors examine the suburban bus route design problem. The objective is to find a shortest tour that visits all the bus stops.

- **Reference:**

Wang, S., & Qu, X. (2014). Rural bus route design problem: model development and case studies. *KSCE Journal of Civil Engineering*, 19(6), 1-5.

Example

- **Problem Description**

Consider a suburban bus route that starts from a bus station, denoted by node 0 and ends at another bus station or city centre denoted by node N+1. Between these two nodes, a bus needs to visit N bus stops. We let 1, 1', 2, 2'...N, N' to represent the bus stops between node 0 and node N+1, where bus stops k and k' correspond to two bus stops at the same geographical location but on both sides of a street, k=1,2…,N. We refer to k as the opposite bus stop of k', and k' as the opposite bus stop of k. We define set K={1,2,…,N}, set K'={1', 2',..., N' }, and set

$$\overline{K} = K \cup K' \cup \{O, N+1\}$$

Example

- **Problem Description**

We let d_{ij} represent the distance between nodes $i \in \overline{K}$ and $j \in \overline{K}$

d_{ij} is a known parameter. We define x_{ij} as the decision variable which equals 1 if and only if the bus visits node $j \in \overline{K}$ immediately after node , and 0 otherwise.

- Mathematical model

$$\min \sum_{i \in \overline{K}} \sum_{j \in \overline{K}} d_{ij} x_{ij}$$

s.t.

$$\sum_{j \in \overline{K}} x_{Oj} = 1$$

$$\sum_{i \in \overline{K}} x_{i,N+1} = 1$$

$$\sum_{i \in \overline{K}} x_{ij} = \sum_{i \in \overline{K}} x_{ij} \quad \forall j \in K \cup K'$$

$$\sum_{i \in \overline{K}} (x_{ik} + x_{ik'}) = 1 \; \forall k \in K$$

$$\sum_{i \in H} \sum_{j \in H} (x_{ij} + x_{ij}) \le |H| - 1, \; \forall H \subseteq \overline{K}$$

$$x_{ij} = \{0,1\} \qquad x_{ii} = 0 \qquad x_{kk'} = 0, x_{k'k} = 0$$

Example

- **Hardness of the Problem**

The decision version of the suburban bus route design problem is in NP, that is, given the bus route it can be determined in polynomial time whether the length of the bus route is shorter than a given constant L. We show the NP hardness of the problem by reducing a well-known NP-complete problem, the Travelling Salesman Problem (TSP), into a suburban bus route design problem.

Example

- **Hardness of the Problem**

Recall that the decision version of the TSP is defined as follows: Given a set of cities $\{0,1,2,\ldots, N\}$, and the distance D_{ij} from city i to j, is there a tour that starts and ends at city 0 and visits each other city exactly once such that the length of the tour is shorter than L?

- **Theorem:**

The decision version of the suburban bus route design problem is NP-hard.

Example

- **Proof:**

Suppose that node 0 and node N+1 coincide. Hence, $d_{i0}=d_{i,N+1}$, $d_{0i}=d_{N+1,i}$, for all $i \in K \cup K'$. Suppose further that node k and node k' coincide for all $k \in K$, that is $d_{ik}=d_{i,k'}$, $d_{ki}=d_{k',i}$, for all $i \in \overline{K}$ and $k \in K$.

It follows easily now that the TSP can be solved by solving a suburban bus route design problem, which means the TSP is reduced to the bus route design problem.

Heuristic Algorithms

Heuristic Algorithms

- **Heuristic algorithm**
  - What is heuristic algorithm?
  - When shall we use heuristic algorithm?
    - P, NP, NP-C, NP-hard
  - Commonly used heuristic algorithms
    - Genetic Algorithm (GA)
    - Artificial Bee Colony (ABC)
    - Simulated Annealing (SA)
    - Tabu Search
    - Artificial Neural Network (ANN)
    - ……

什么是启发式算法

- 启发式算法（heuristic algorithm)是相对于最优化算法（exact algorithm）提出的。
- 一个问题的最优算法可求得该问题的最优解。
- 启发式算法可以这样定义：一个基于直观或经验构造的算法，在可接受的花费（指计算时间和空间）下给出待解决组合优化问题每一个实例的一个可行解，该可行解与最优解的偏离程度一般不能被预计。（除非可以提供 Gap Function）

Heuristic Algorithms

- A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution.
- The purpose of heuristic is to obtain a solution that is near optimal, or good, or at least better than an arbitrary solution, in a reasonable time.

Metaheuristic Algorithms

- A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the ***search space***. Learning strategies are used to structure information in order to find efficiently near-optimal solutions.
- **Example**
  - Particle Swarm Optimization
  - Artificial Bee Colony
  - Ant Colony Optimization.

Limitations of Heuristic Algorithms

- Cannot guarantee the optimality of final results.
- Some times even inefficient than an enumeration method.
- If a problem with restrictive constraints, it is difficult to generate feasible solutions.

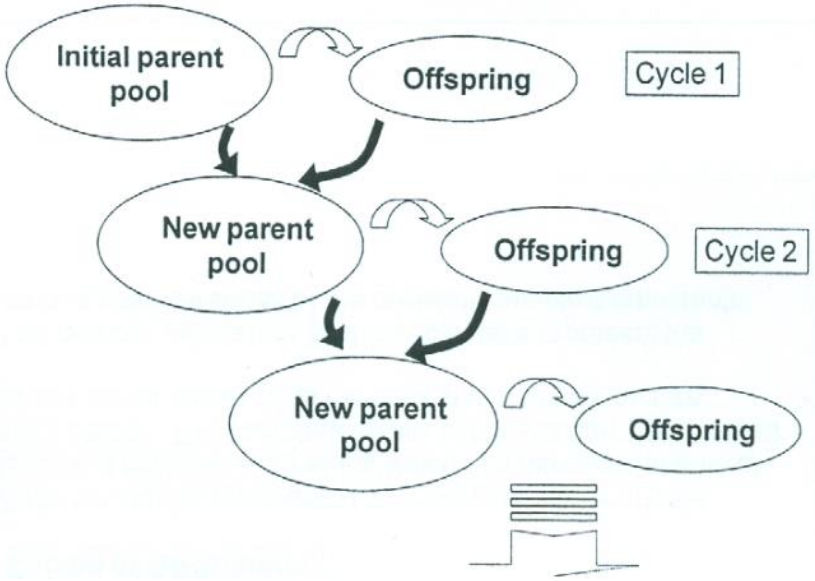
Heuristic Algorithms

- **When shall we use Heuristic algorithm?**
  - When it's too time consuming to solve a problem using conventional exact algorithms.
- **A quantitative measure of "time consuming".**
- **Time complexity → NP-hardness**
  - P: Polynomial:多项式(时间)
  - NP: Non-deterministic polynomial: 非确定性多项式(时间)
  - NP-Complete (NPC)
  - NP-hard

Genetic Algorithm

Genetic Algorithm

- Definition
  - In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.



Genetic Algorithm

- Coding of GA
  - Each solution is represented by a string structure of chromosomes. Each cell contains a numerical value of a decision variable (gene) of the problem.
  - The complete details for the solution are revealed by the coded values of all the genes of the chromosomes.
  - The feasible values that each cell can take represent the various possible states of the decision variable it represents.
- An example for chromosome

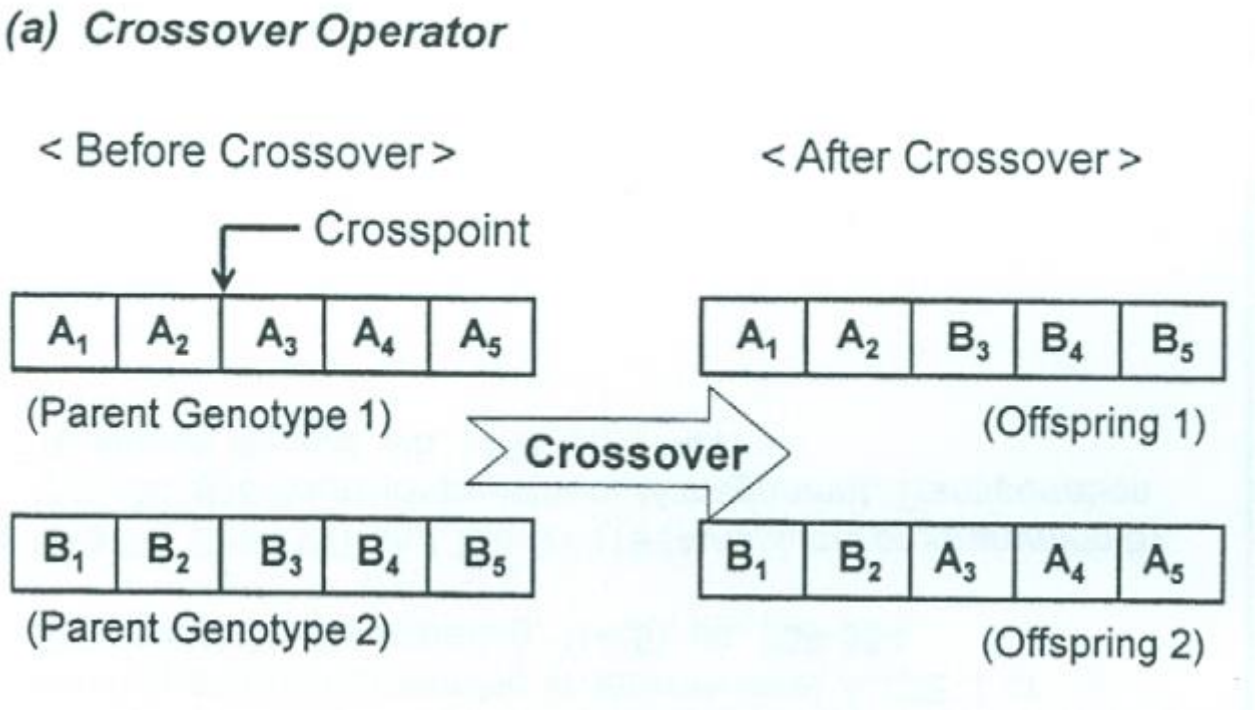
τ_1	τ_2	.....	τ_i	τ_{i+1}	.....	τ_{n-1}	τ_n

Genetic Algorithm

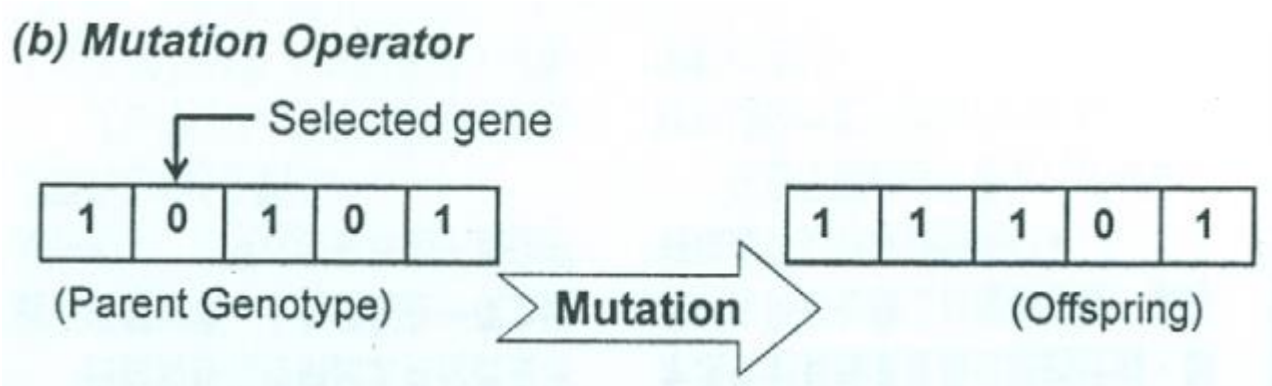
- Key factors in GA
  - Population size
  - Initial population
    - Generated by random numbers (satisfying all constraints)
  - Evaluation of each chromosome (Fitness function)
  - Generating new chromosome
    - Crossover
    - Mutation
  - Evaluation of new chromosome
  - Convergence Criteria

Genetic Algorithm

- Crossover:



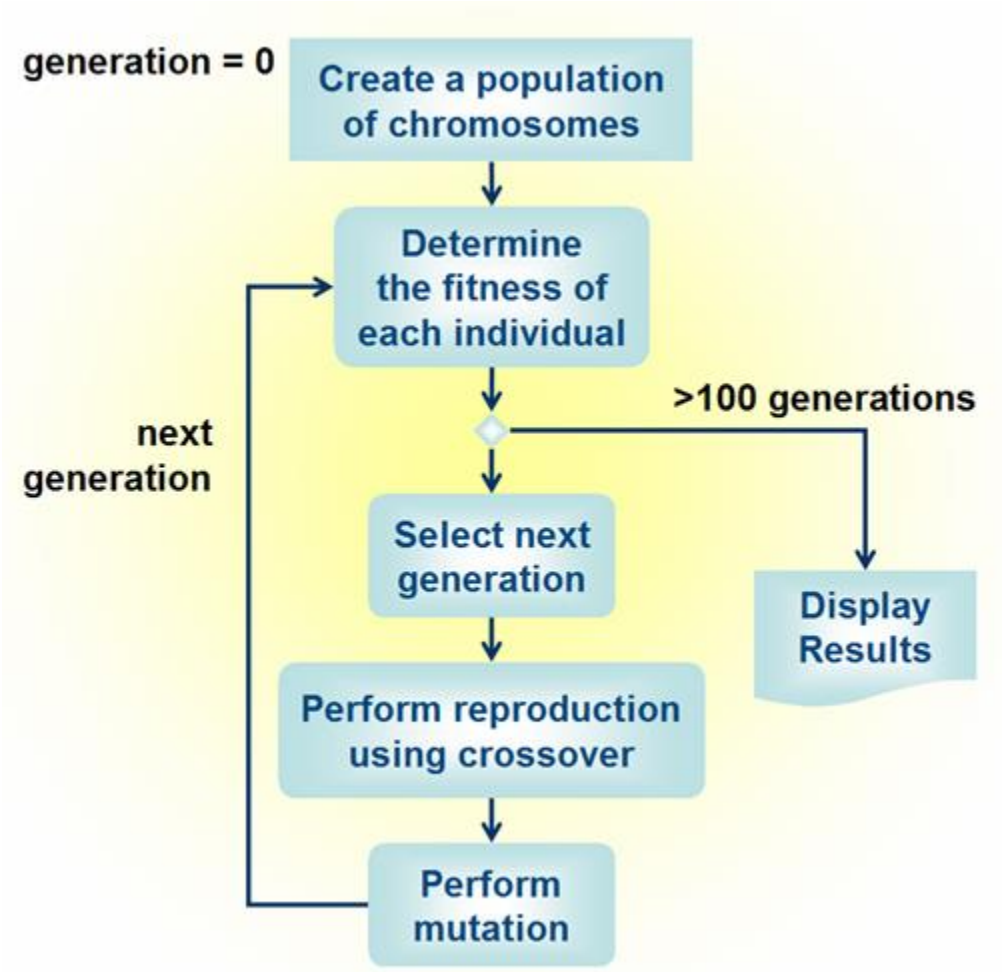
- Mutation:



Genetic Algorithm

- Fitness Evaluation: based on user-defined objective function.
- Examples of objective function
  - Maximize production
  - Minimize cost
  - Maximize utilization of manpower
  - Maximize utilization of equipment
  - Maximize utilization of allocated budget
  - Maximize network pavement condition
  - Maximize pavement service lives

Genetic Algorithm



Genetic Algorithm

- Convergence Criteria
  - Predetermined generation number
  - Best-solution performance – compare the best solution of each generation
  - Offline parent-pool performance – compare average fitness of each parent pool
  - Offline offspring-pool performance – compare average fitness of each offspring pool
  - Online performance – monitor the running average of all valid genotypes that have been generated

Genetic Algorithm

- **Step 1**: (Initial population). Set the size of population to be $\bar{n}$. Randomly generate initial population of the chromosomes, which contains toll fares on each tolled link. Let number of generation $m = 1$.
- **Step 2**: (Crossover). Randomly choose some parents from the survivors, and conduct pairing between each parent, which yields some new chromosomes.
- **Step 3: (**Mutation). With a lower probability, randomly choose some genes from all the chromosomes in current generation, and then modify value of these genes by a pseudo random number between 0 and τ^{max}. This process also generates some new chromosomes.

Genetic Algorithm

- **Step 4**: (Evaluation). For each newly generated chromosome, solve a traffic assignment using the Frank-Wolfe method, and then record its total travel time.
- **Step 5**: (Selection). Among all the existing individuals, choose the top individuals with fewer total travel times, as survivors for next generation.
- **Step 6**: (Stop Test). If $m > m_{\max}$, then stop, where $m_{\max}$ is a predetermined upper-bound for the number of generations; otherwise, set $m = m + 1$ and go to Step 2.

Example 1

$$\min f(x) = (x_1 - 5x_2)^3 - \exp(x_2 \times x_3) + 7x_1$$

$$-20 \le x_i \le 100, i = 1, 2, 3$$

	x_1	x_2	x_3
Genes1	3.7	2.0	10.2
Genes2	15.7	8.6	0.5
Genes3	-5.3	15.6	34.1

……（100 genes in total）

Randomly choose Genes2、3

CROSSOVER

Choose cross point randomly

Genes2	15.7	8.6	0.5
Genes3	-5.3	15.6	34.1

Offspring

Genes2’	15.7	15.6	34.1
Genes3’	-5.3	8.6	0.5

CROSSOVER

Example 1

Genes5	14.3	2.4	4.5
Genes51	-5.2	10.6	3.1
Genes76	14.2	1.6	35.6
Genes92	-3.3	7.6	7.5

MUTATION

Randomly choose some genes (from all the chromosomes in current generation

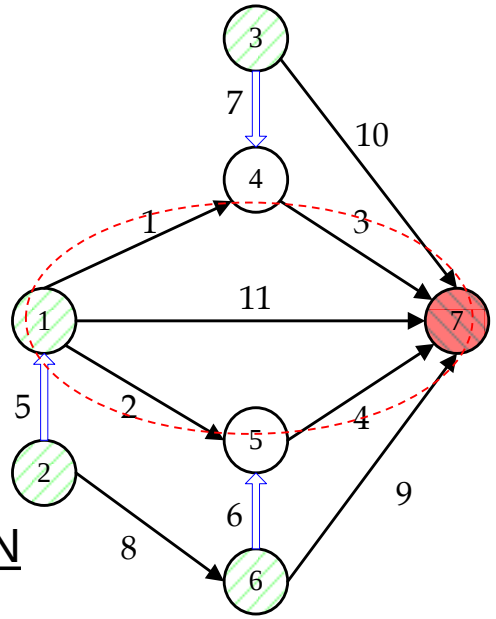
Genes5'	14.3	2.8	4.5
Genes51'	-6.1	10.6	3.1
Genes76'	13.8	1.6	35.6
Genes92'	-3.3	7.6	8.5

Offspring

Example 2: Road Pricing on Entry Links

- **A feasible chromosome:**

Tolled Link	5	6	7
Chromosome 1: Genes	3.7	2.0	10.2
Chromosome 2: Genes	15.7	8.6	0.5



CROSSOVER

MUTATION

Genes	13.3	7.3	2.4
Genes	6.1	3.3	8.3

Genes	15.7	11.6	0.5

$$\hat{Y}_1 = \chi \bar{Y}_1 + (1-\chi)\bar{Y}_2$$
$$\hat{Y}_2 = \chi \bar{Y}_2 + (1-\chi)\bar{Y}_1$$
$$\chi \in (0,1), \chi \neq 0.5$$

Generating Random Numbers

- Generation of random numbers uniformly distributed between 0 and 1.
  - These can be broadly classified into *manual* and *mathematically-based* techniques.
  - Manual techniques involve the development of random digits from published sources of random numbers, using the last four digits from telephone numbers etc.
- "Pseudo" random numbers
  - Generated by computer program.
  - Linear functions with large parameters are usually used.
  - This is how random numbers are generated in all statistical/ mathematical software including Excel.

Linear Congruential Method

- "Pseudo" random numbers
  - Linear congruential method (线性同余法)

$$S_i = [a\ S_{i-1} + b] \bmod c$$

$$R_i = S_i / c$$

S_i is random series (S_0 is random seed), R_i is random numbers ($0 < R_i < 1$), a is multiplier ($0 < a < c$), b is increment ($0 < b < c$), c is modulus ($c > 0$),

Cycle: length of series before numbers repeat. [Full cycle = all numbers represented]

Linear Congruential Method

$$S_i = [a\ S_{i-1} + b] \bmod c$$

Suppose a=7, b=5, c= 12, S_0=4.

S	4	9	8	1	0	5	4
a S + b	33	68	61	12	5	40	33

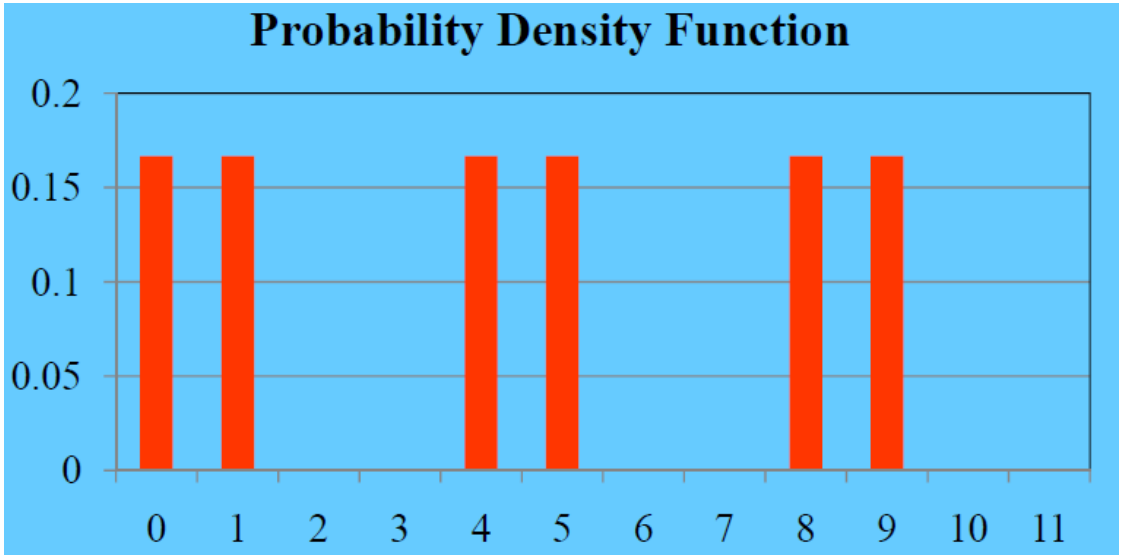
- Notice max value is 9; cycle length=6
- Only ½ cycle since theoretical max number is 12
- Suppose Random seed is 2, a different series results

S	2	7	6	11	10	3	2
a S + b	19	54	47	82	75	26	19

Linear Congruential Method

Missing Numbers and Imperfect resolution

S	4	9	8	1	0	5	4
a S + b	33	68	61	12	5	40	33



Generating Random Numbers

How Does Excel Generate Random Numbers?

Excel generates random numbers as follows:

- First random number (R1) = fractional part of (9821 * r + 0.211327), where r = .5
- Second random number = fractional part of (9821 * R1 + 0.211327).

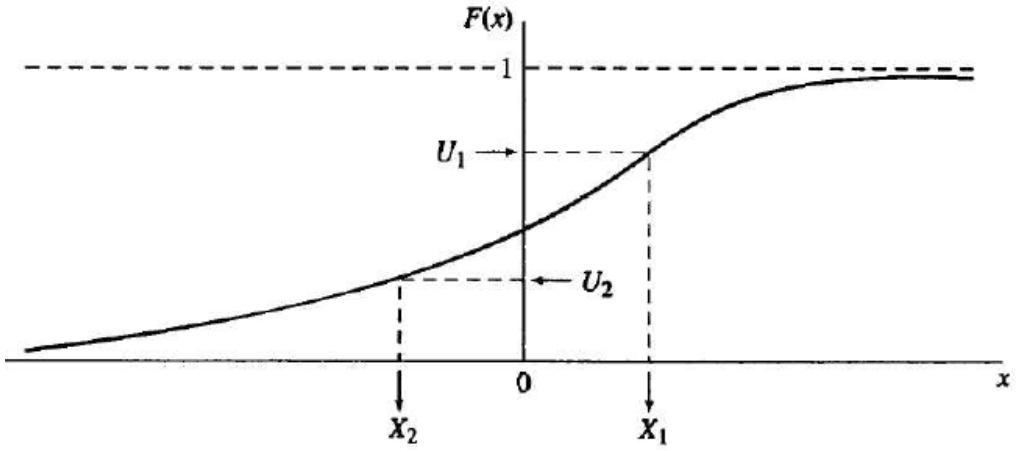
The implications are that you can generate up to 1,000,000 random numbers. Many have questioned the 'randomness' of these numbers and several free downloads of random number generators are available.

To generate random numbers in Excel use the RAND () function. Note, that this will re-generate random numbers every time you perform a calculation. You should save your numbers with 'Copy-Paste-Special-Values' to avoid this happening.

Think about how to generate different set of numbers every time?

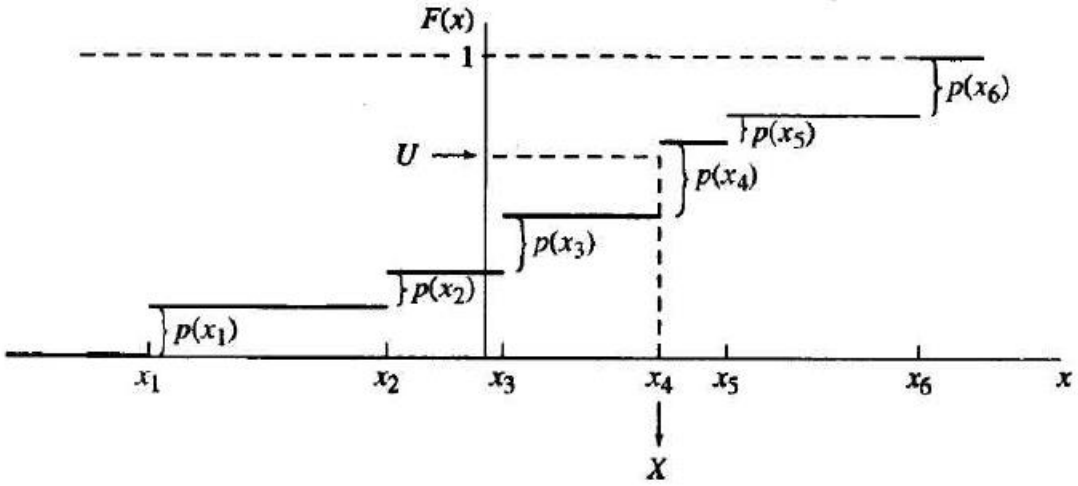
Generating Random Numbers

- Generation of random numbers for other distributions.
  - Using the Cumulative Distribution Function
- 1. Continuous random variable:



Generating Random Numbers

- 2. Discrete Random Variable



Generating Random Numbers

- Summary for "Pseudo" random numbers
  - The algorithms vary in their complexity and the 'randomness' of the numbers produced
  - Eventually, the pseudo-random numbers will repeat themselves, but this is typically after generating a very large number, such as 10^9
  - The algorithms are initiated based on an initial number (*seed*). When the seed is changed, a different set of pseudo-random numbers will result. More significantly, by keeping the seed the same, the user knows the numbers will be the same, which is very useful when developing/testing simulation applications

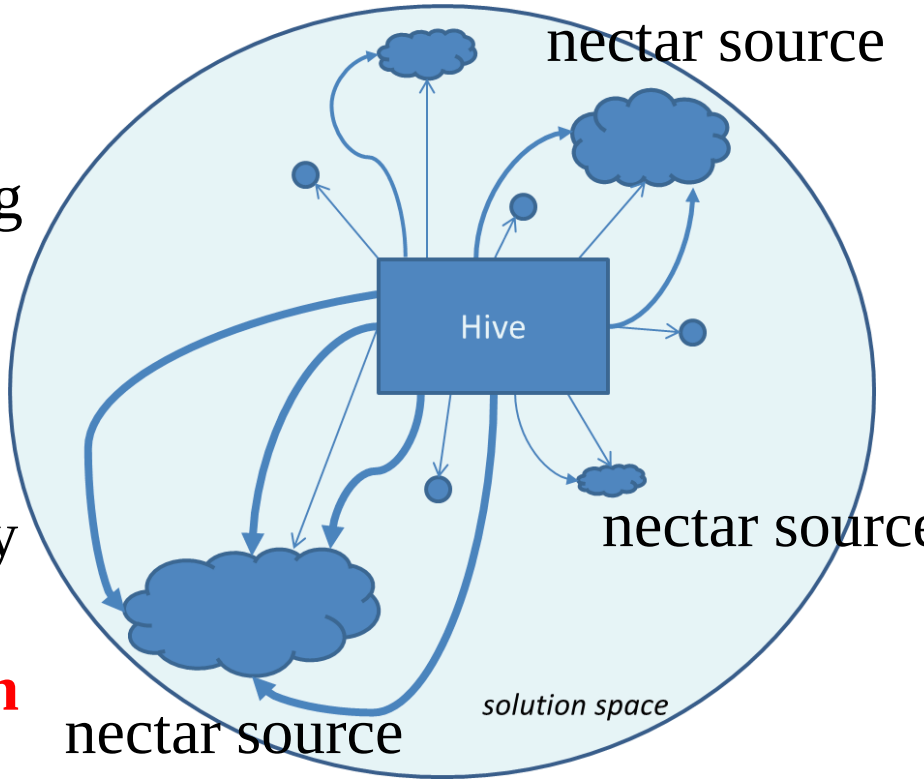
Artificial Bee Colony (ABC) algorithm

Artificial Bee Colony (ABC) algorithm

❑ Overview of ABC algorithm

- The ABC algorithm is a **population-based** metaheuristic approach motivated by the foraging behavior of honey bees finding nectar sources around the hive.

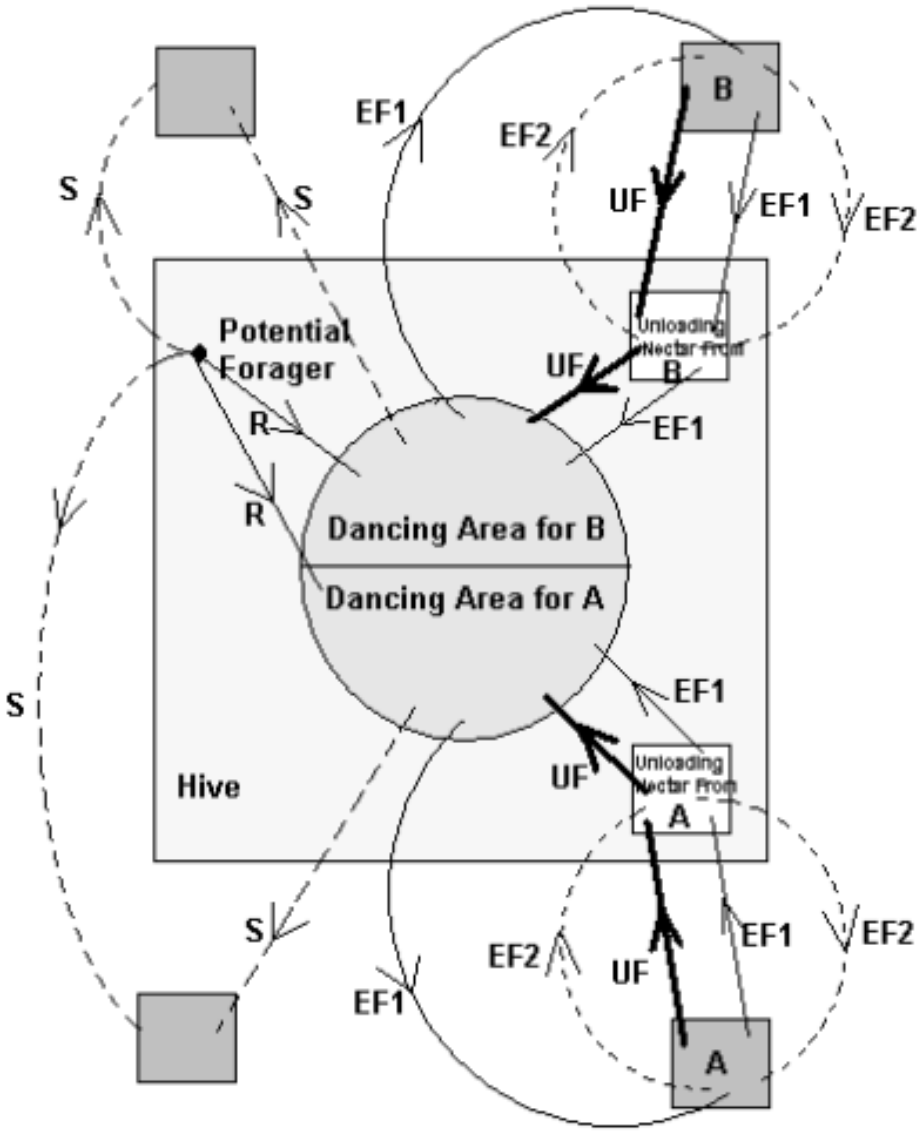
- Compared with extant evolutionary algorithms like GA, the ABC algorithm has a **better local search mechanism** that enhances the solution quality.



Foraging Behavior

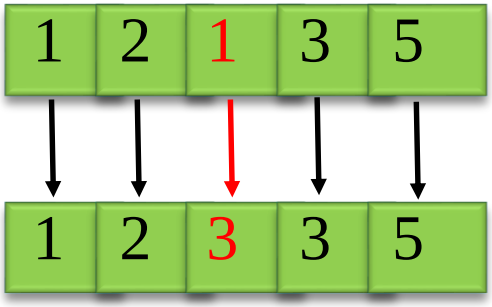
- Types of foraging bee
  - Employed bees (雇佣蜂)
  - Unemployed bees (非雇佣蜂)
    - Scout (侦查蜂)
    - Onlooker bees (观察蜂)

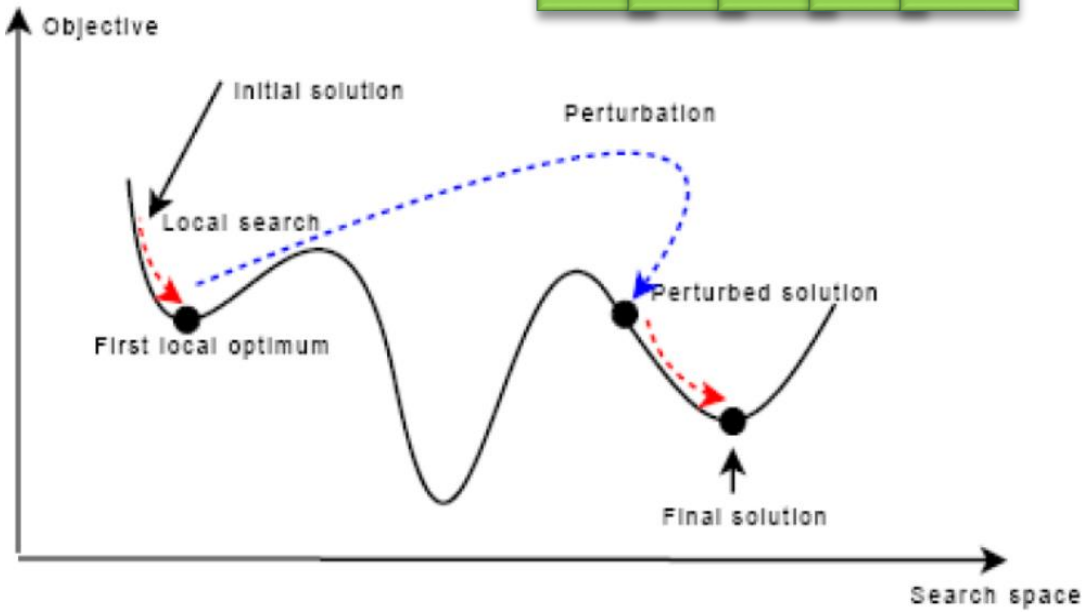
Scouts are to explore randomly new sources of food. They explore local neighborhood of known food sources. The information about the food source and amount of food it posses is shared in the hive. The more food contains the source the higher its chance to be picked up as target to fly to.



Foraging Behavior

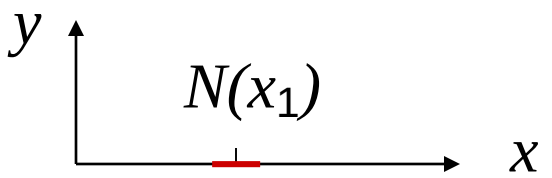
- Compared with extant evolutionary algorithms like GA, the ABC algorithm has a better ***local search mechanism*** that enhances the solution quality

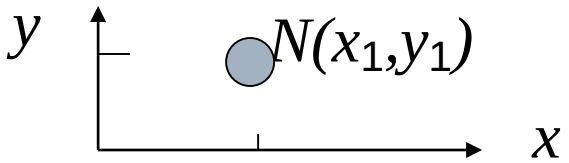




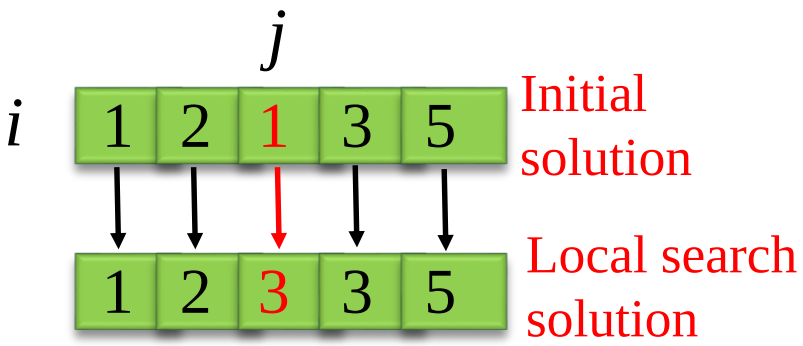
Local Search Mechanism

- In a one-dimensional space
  - The neighborhood is a small **interval** around the point x_1 .
- In a two-dimensional space
  - The neighborhood is a small **area** around the point (x_1, y_1).





- Example: Perform a local search for the *i* th solution



Firstly, choose the *j* th parameter randomly. Then,

$$x_{ij}^{New} = x_{\min,j} + rand(0,1) \times (x_{\max,j} - x_{\min,j})$$

where

$x_{\max,j}, x_{\min,j}$ is the upper- and lower- bound of x_{ij}

ABC Algorithm

***Step 1*:** Initialize the parameters

Colony size: N_c

No. of employed bees: N_e

No. of onlookers: N_o

No. of scouts: N_s

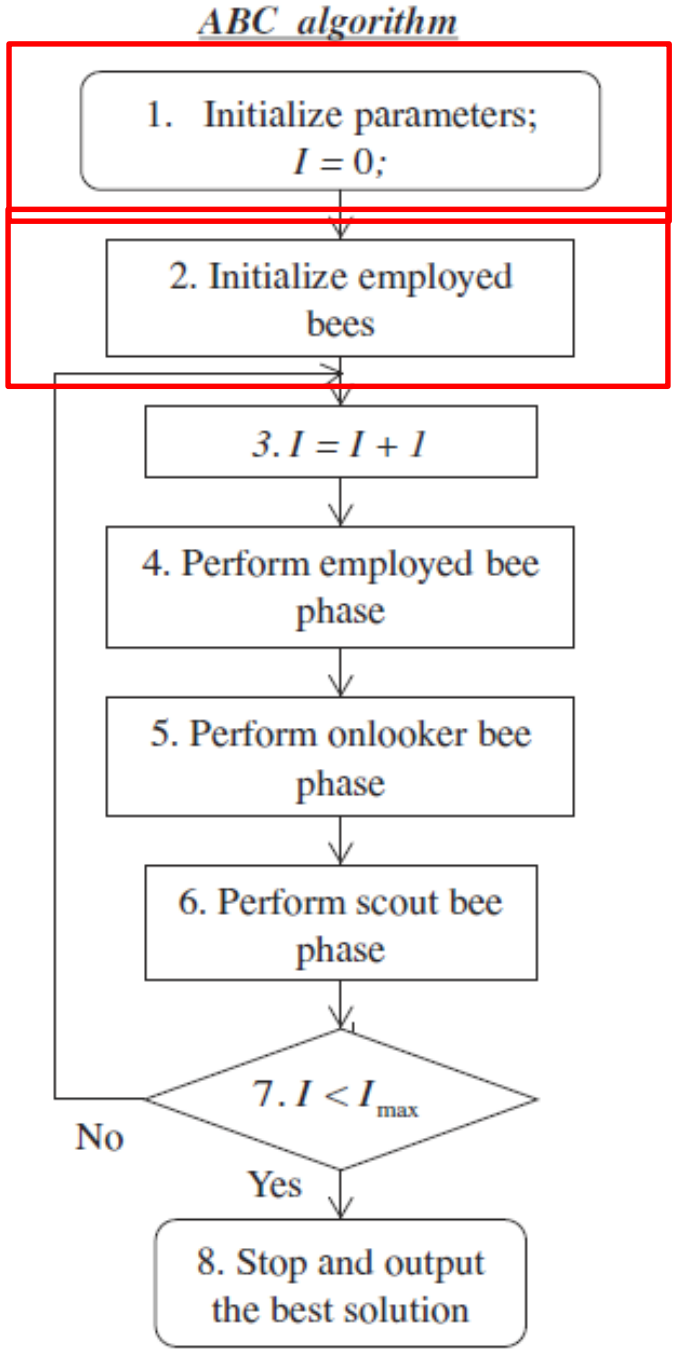
Limits:

Counter of iterations: I

Maximum number of iterations: I_{max}

***Step 2*:** Initialization of employed bees

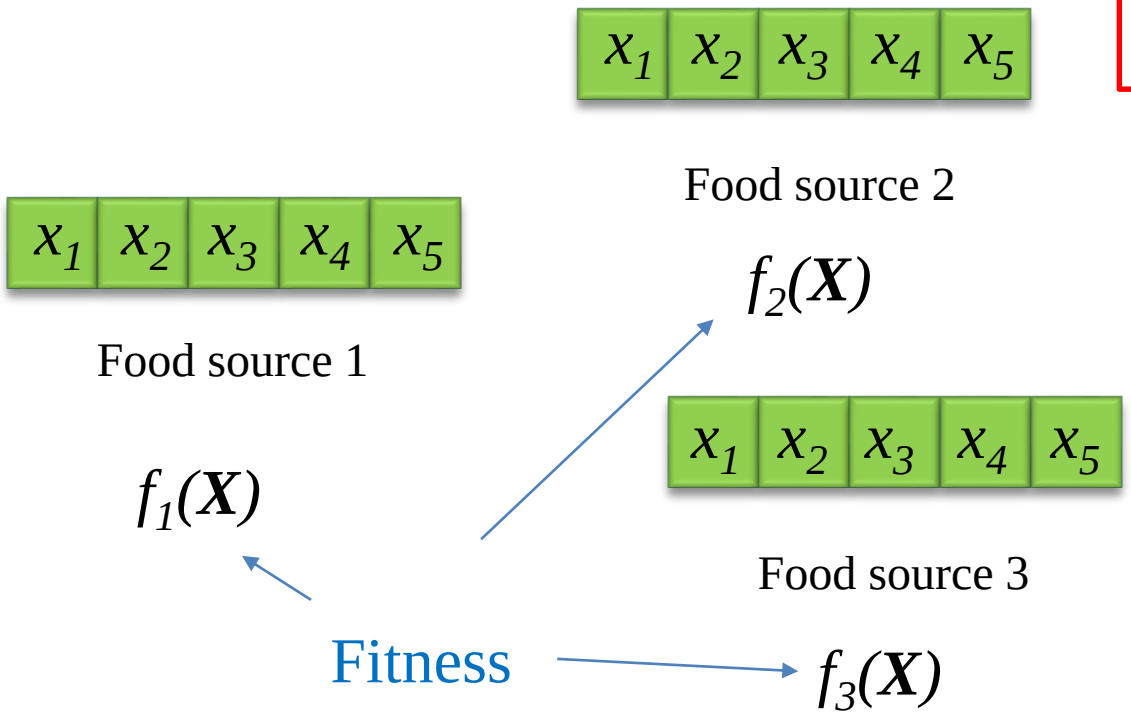
Generate an initial food source (solution) for each employed bee and the limit counter for each food source is set to be zero.

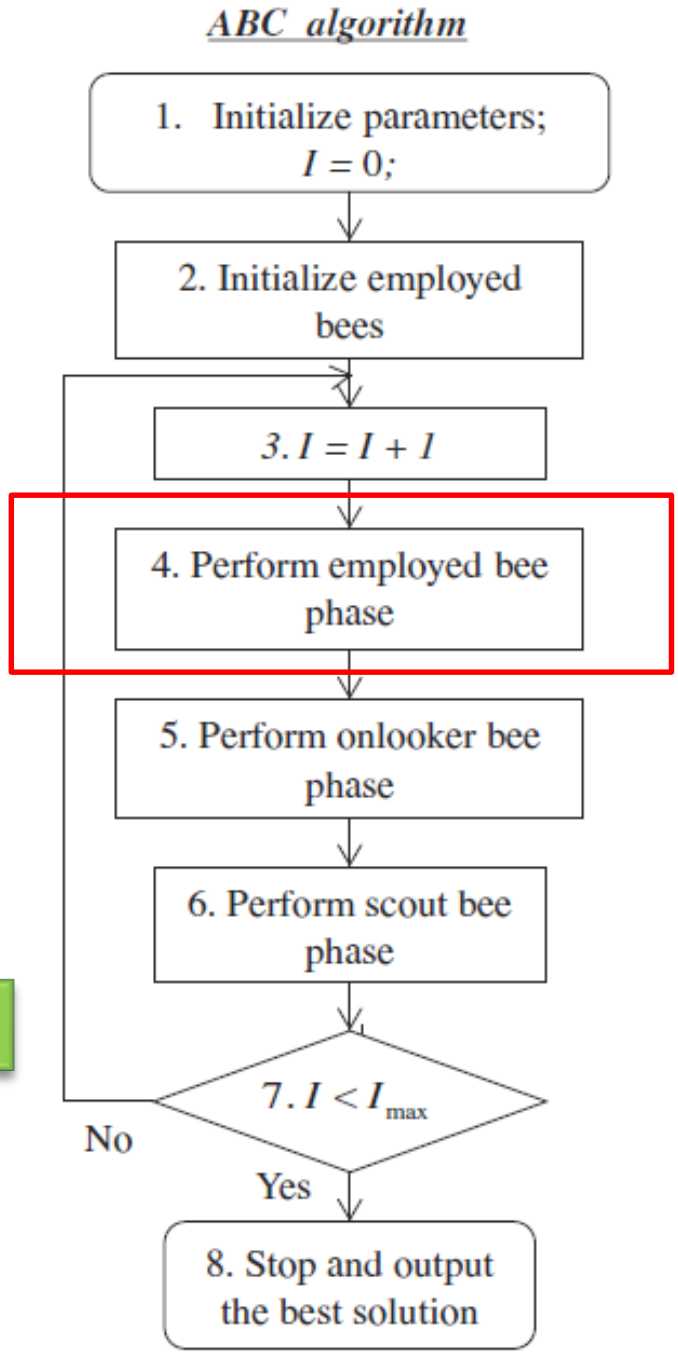


ABC Algorithm (con't)

***Step 3*:** Perform the employed bee phase

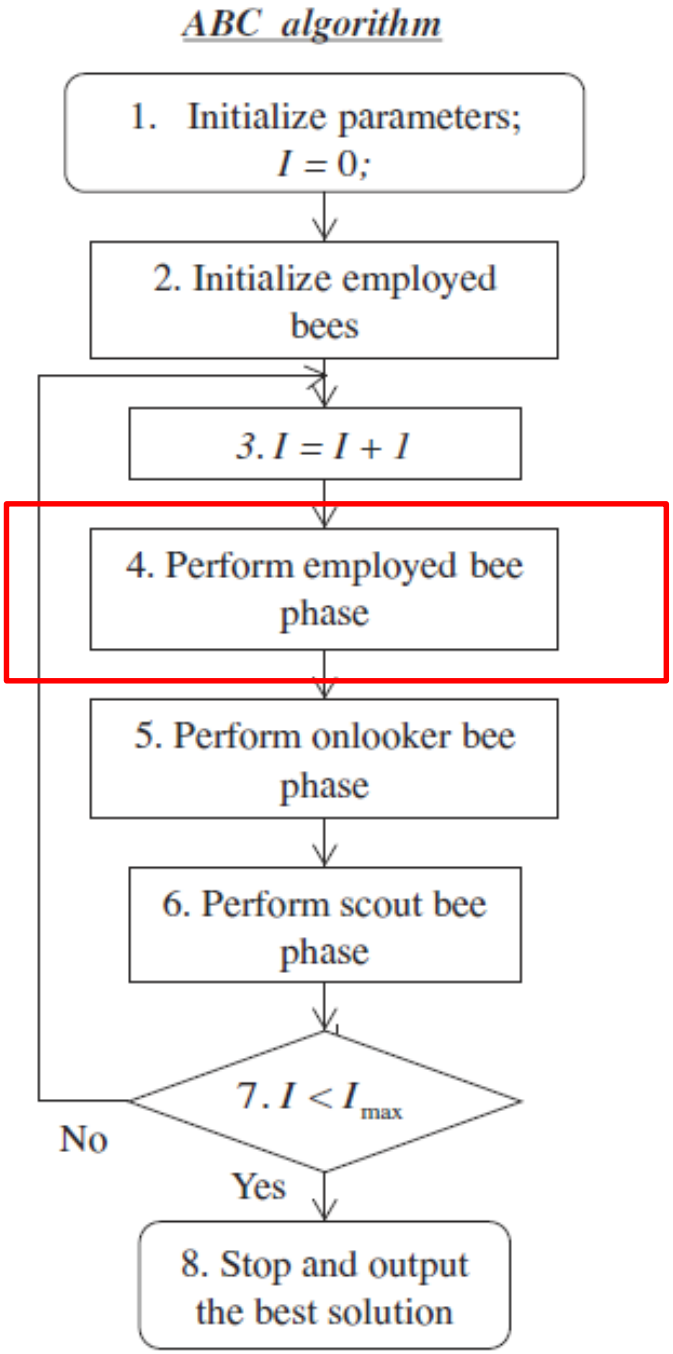
- Number of food source (solutions) = Employed bees
- Number of genes in a food source =number of variables





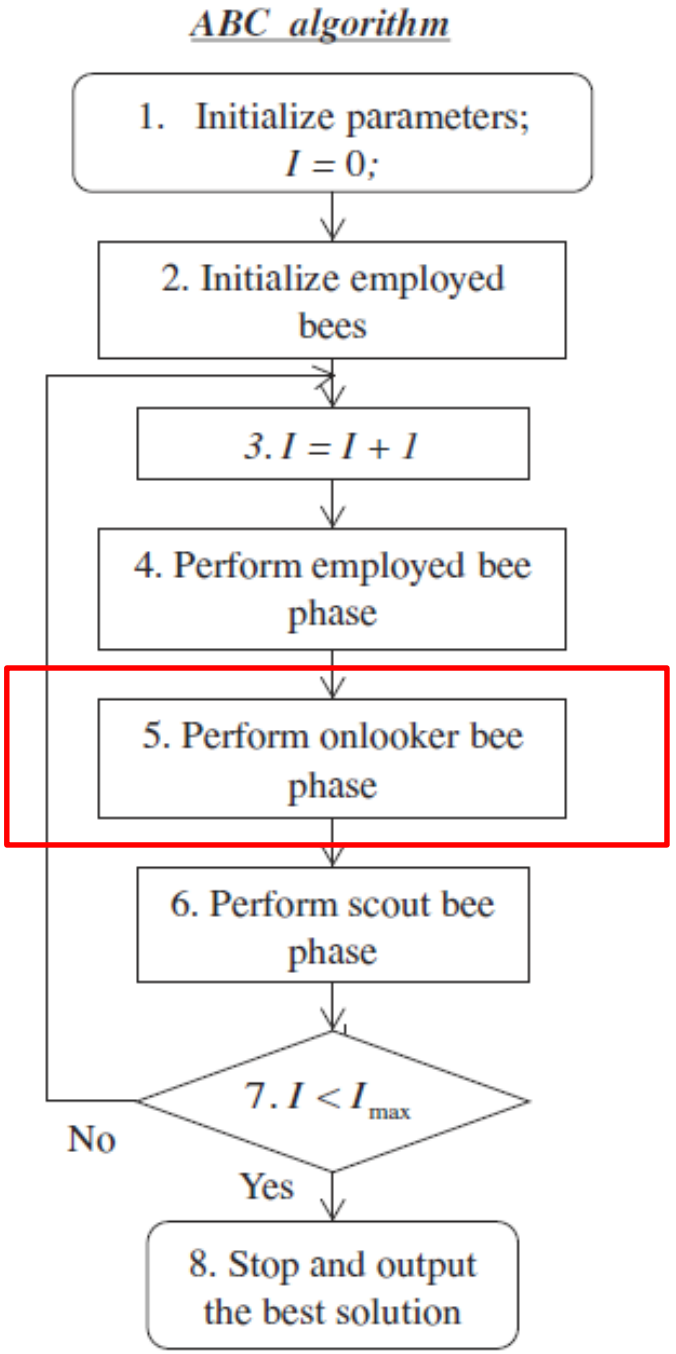
ABC Algorithm (con't)

***Step 3*:** Perform the employed bee phase Execute a neighborhood search for each food source found by an employed bee. Evaluate the fitness of each neighbor solution. If the fitness of one neighbor solution is better than the former solution, replace the solution by this neighbor solution and set its limit counter to zero; otherwise, keep the former solution of the employed bee, and increase the corresponding limit counter by 1.



ABC Algorithm (con't)

***Step 4*:** Perform the onlooker bee phase Conduct the roulette wheel selection to determine which food source obtained by an employed bee is chosen by an onlooker. Then, execute a neighborhood search for each solution selected by an onlooker. Evaluate the fitness of each neighbor solution. Replace the solution by its neighbor solution, if the fitness of this neighbor solution is better. Otherwise, keep the former solution of the employed bee, and increase its limit counter by 1.



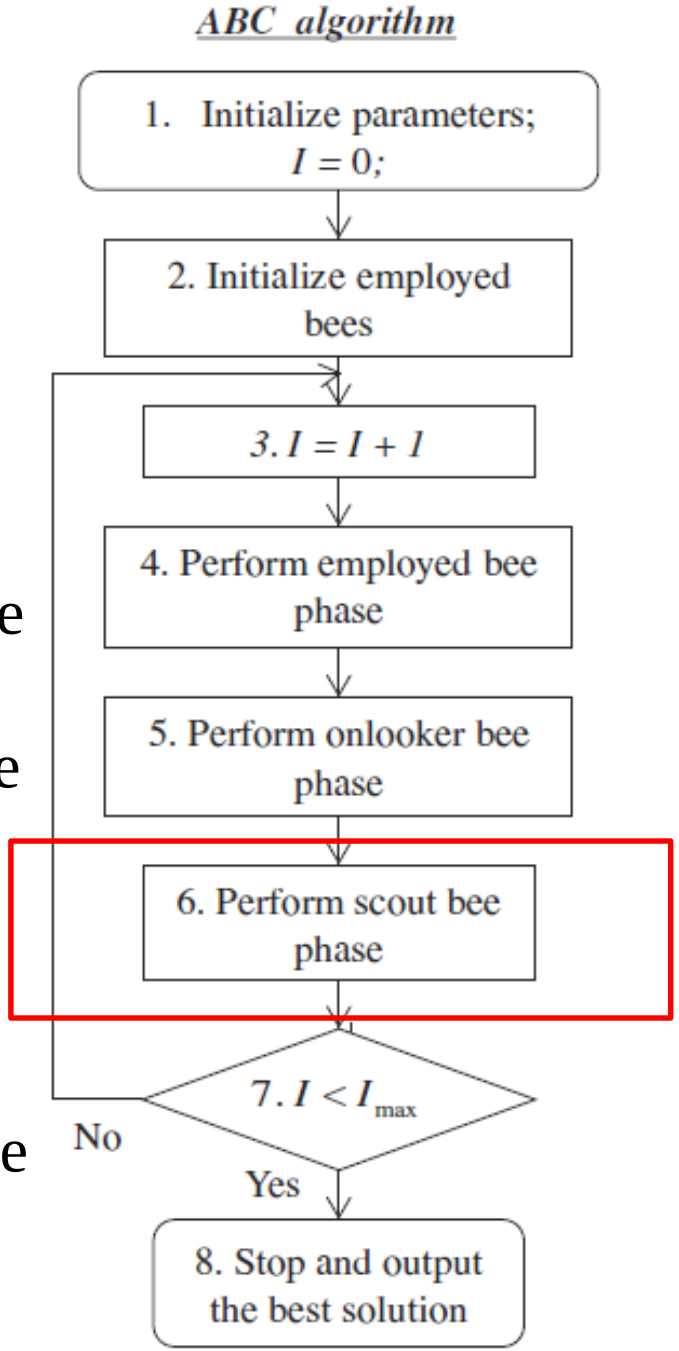
ABC Algorithm (con't)

***Step 5*:** Perform the scout bee phase

Compare the fitness of all food sources obtained by N_e employed bees, and find and keep the best food source with the highest fitness.

If one food source cannot improve within the maximal trial number limit, and meantime it is not the best food source, the employed bee belonging to this food source abandon this food source and generate a new food source randomly. Set the limit counter of this food source to zero.

Notes: if the best food source has reached the maximum limit, it would not be abandoned.

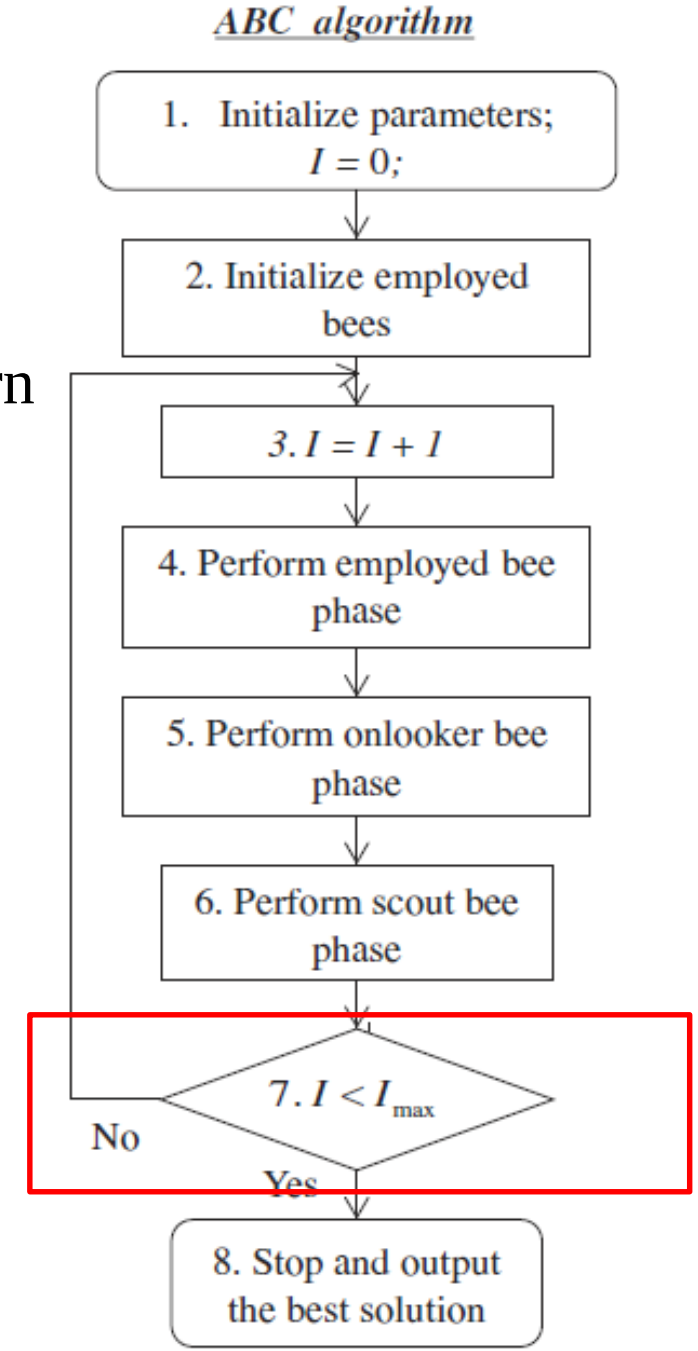


ABC Algorithm (con't)

***Step 6*:** Stop test

Increase the number of iterations by 1.

Check the stopping criterion: if $I<I_{max}$, return to ***Step 3***; otherwise, terminate and output the best solution.



Advantages & Disadvantages

- Advantages
  - Few control parameters
  - Fast convergence
  - Both exploration & exploitation
- Disadvantages
  - Search space limited by initial solution (normal distribution sample should use in initialize step)

Example (Designed by Qixiu Cheng)

$$\min z(\mathbf{x}) = \sum_{i=1}^{2} x_i^2$$

$$s.t. \ -10 \le x_i \le 10$$

Step 1: Initialize the parameters

For simplicity, we set the colony size $N_c = 5$, the number of employed bees $N_e = 3$, onlookers $N_o = 1$, scouts $N_s = 1$; the predetermined number of iterations $LC_{max} = 3$; the initial value of iteration counter $I = 1$, and its maximum value $I_{max} = 100$.

Fitness Function: $f(x_i) = 1/[1 + z(x_i)]$

Example

Step 2: Initialization of employed bees

The generated initial food sources are as follows:

$$\mathbf{x}_1^{(0)} = (x_1, x_2)^T = (3, -1)^T$$
$$\mathbf{x}_2^{(0)} = (x_1, x_2)^T = (6, -8)^T$$
$$\mathbf{x}_3^{(0)} = (x_1, x_2)^T = (-2, 5)^T$$

The fitness of the current food source:

$$f^{(0)}(\mathbf{x}_1) = 1/(1+10) = 0.091$$
$$f^{(0)}(\mathbf{x}_2) = 1/(1+100) = 0.009$$
$$f^{(0)}(\mathbf{x}_3) = 1/(1+29) = 0.033$$

Set the limit counter of each food source be zero

$$LC_1^{(0)} = LC_2^{(0)} = LC_3^{(0)} = 0.$$

Example

Iteration Procedure

Step 3: Employed bee phase

Assume that the generated neighbor solutions are:

$$\mathbf{x}_1^{(1)} = (x_1, x_2)^T = (3, 0)^T$$
$$\mathbf{x}_2^{(1)} = (x_1, x_2)^T = (6, -7)^T$$
$$\mathbf{x}_3^{(1)} = (x_1, x_2)^T = (-2, 6)^T$$

Evaluate the corresponding fitness of neighbor solutions:

$$f^{(1)}(\mathbf{x}_1) = 1/(1+9) = 0.100$$
$$f^{(1)}(\mathbf{x}_2) = 1/(1+85) = 0.012$$
$$f^{(1)}(\mathbf{x}_3) = 1/(1+40) = 0.024$$

Example

Step 3: Employed bee phase

Compare the fitness between the current food sources and the neighbor solutions:

$$f^{(1)}(\mathbf{x}_1) > f^{(0)}(\mathbf{x}_1) \qquad f^{(1)}(\mathbf{x}_2) > f^{(0)}(\mathbf{x}_2) \qquad f^{(1)}(\mathbf{x}_3) < f^{(0)}(\mathbf{x}_3)$$

From the comparison above, we can find that the first two neighbor solutions are better than the current food sources, so we replace the current food sources by the neighbor solutions and set the limit counter of the first two food sources be 0, i.e., $LC_1^{(1)} = LC_2^{(1)} = 0 < LC_{max}$. However, the last neighbor solution is worse than the current food source, so we keep the current food source unchanged and now increase the limit counter by 1, i.e., $LC_3^{(1)} = 1 < LC_{max}$.

Example

Step 3: Employed bee phase

Now the new food sources are:

$$\mathbf{x}_1^{(1)\prime} = (x_1, x_2)^T = (3, 0)^T$$

$$\mathbf{x}_2^{(1)\prime} = (x_1, x_2)^T = (6, -7)^T$$

$$\mathbf{x}_3^{(1)\prime} = (x_1, x_2)^T = (-2, 5)^T$$

Calculate the corresponding fitness:

$$f^{(1)\prime}(\mathbf{x}_1) = 1/(1+4) = 0.100$$

$$f^{(1)\prime}(\mathbf{x}_2) = 1/(1+85) = 0.012$$

$$f^{(1)\prime}(\mathbf{x}_3) = 1/(1+29) = 0.033$$

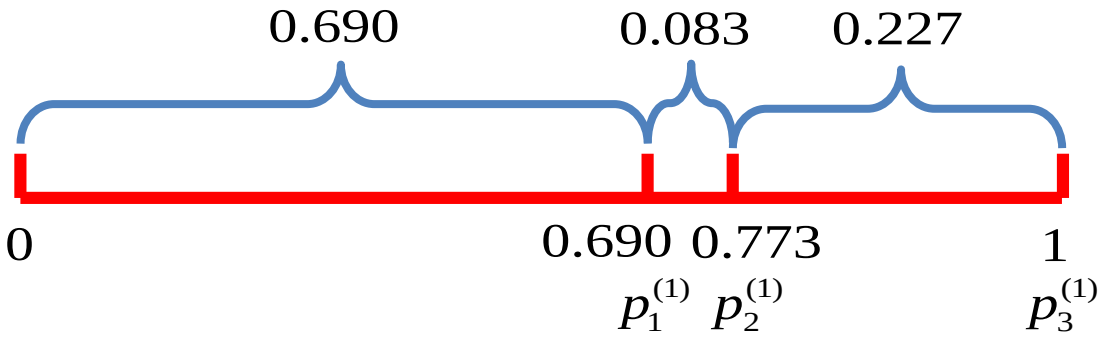
Step 4: Onlooker phase

In this example, the probability of each food source which should be followed by onlookers is

$$p_1^{(1)} = f^{(1)\prime}(\mathbf{x}_1) / \sum_{i=1}^{3} f^{(1)\prime}(\mathbf{x}_i) = 0.100 / (0.100 + 0.012 + 0.033) = 0.690$$

$$p_2^{(1)} = f^{(1)\prime}(\mathbf{x}_2) / \sum_{i=1}^{3} f^{(1)\prime}(\mathbf{x}_i) = 0.012 / (0.100 + 0.012 + 0.033) = 0.083$$

$$p_3^{(1)} = f^{(1)\prime}(\mathbf{x}_3) / \sum_{i=1}^{3} f^{(1)\prime}(\mathbf{x}_i) = 0.033 / (0.100 + 0.012 + 0.033) = 0.227$$



Generate random number: $r = 0.843$

The third onlooker will be executed a local search, others keep unchanged

Example

Step 4: Onlooker phase

Assume that the neighbor food sources generated by the third onlookers are:

$$\mathbf{x}_3^{(1)''} = (x_1, x_2)^T = (-1, 5)^T$$

Calculate the corresponding fitness:

$$f^{(1)''}(\mathbf{x}_3) = 1/(1+26) = 0.037$$

Compare the fitness between the current food sources and the neighbor food sources:

$$f^{(1)''}(\mathbf{x}_3) > f^{(0)'}(\mathbf{x}_3)$$

Example

Step 4: Onlooker phase

From these comparisons, we can find that the last neighbor solution is better than the current food source, so we replace the current food source by the neighbor solution. And set $LC_3^{(1)} = 0$.

However, the first and second neighbor solutions is worse than the current food sources, so we keep the current food source unchanged and increase the limit counter by 1, i.e., $LC_1^{(1)} = 1, \quad LC_2^{(1)} = 1.$

$$LC_1^{(1)} < LC_{\max}, LC_2^{(1)} < LC_{\max}, LC_3^{(1)} < LC_{\max}$$

Example

Step 5: Scout bee phase

The current food sources are:

$$\mathbf{x}_1^{(1)'''} = (x_1, x_2)^T = (3, 0)^T$$

$$\mathbf{x}_2^{(1)'''} = (x_1, x_2)^T = (6, -7)^T$$

$$\mathbf{x}_3^{(1)'''} = (x_1, x_2)^T = (-1, 5)^T$$

Calculate the corresponding fitness:

$$f^{(1)'''}(\mathbf{x}_1) = 1/(1+9) = 0.100$$

$$f^{(1)'''}(\mathbf{x}_2) = 1/(1+85) = 0.012$$

$$f^{(1)'''}(\mathbf{x}_3) = 1/(1+26) = 0.037$$

Example

Step 5: Scout bee phase

- Note that the first food source is the best one with the highest fitness. No food source has executed the predetermined maximal trial number, so no food source will be exhausted in this iteration procedure.

Example

Step 6: Convergence test

In this iteration procedure, $I = 1$, $I_{max} = 100$, $I < I_{max}$, so we continue the iteration procedure until it reaches the stop condition.

Number	x_1	x_2	$f(\mathbf{x})$
1	0.01395740875923565	-0.07851484449200902	0.006359390064876747
2	-0.23964187657939306	-0.002223738299727565	0.05743317402251873
3	0.008622216326064934	0.043272436322771696	0.001946846359681792
4	-0.1488691692153740 8	-0.12998682884249477	0.03905860521540371
5	-0.12123014382342268	0.05440253782761567	0.0176563838935329
6	-0.1813243823970262	-0.021012929363252275	0.03332007485208801
7	0.04540570116463309	0.03219740042774366	0.00309835029255643
8	-0.17449297079646753	-0.1484896707703789	0.05249697918287238
9	0.03988346849051805	0.09245947375260188	0.010139445345442222
10	0.03878208764555202	-0.10914980121196982	0.013417729426759808