

人工智能 高级搜索

陈川

中山大学 计算机学院

2024年



中山大學
SUN YAT-SEN UNIVERSITY

高级搜索

- 爬山算法
- 模拟退火算法
- 遗传算法
- 应用案例

局部搜索算法

- 搜索算法在内存中保留一条或多条路径并且记录哪些是已经探索过的，哪些是还没有探索过的。当找到目标时，到达目标的路径同时也构成了这个问题的一个解。
 - ◆ 在许多问题中，问题的解与到达目标的路径是无关的。
 - ◆ 例如，在八皇后问题中，重要的是最终皇后的布局，而不是加入皇后的次序。
 - ◆ 这类问题包括了：
集成电路设计；工厂场地布局；作业车间调度；
自动程序设计；电信网络优化；车辆寻径；文件夹管理
- 如果目标路径与问题解不相关，将考虑各种根本不关心路径（耗散）的算法，其中占据重要地位的就是**局部搜索算法**；

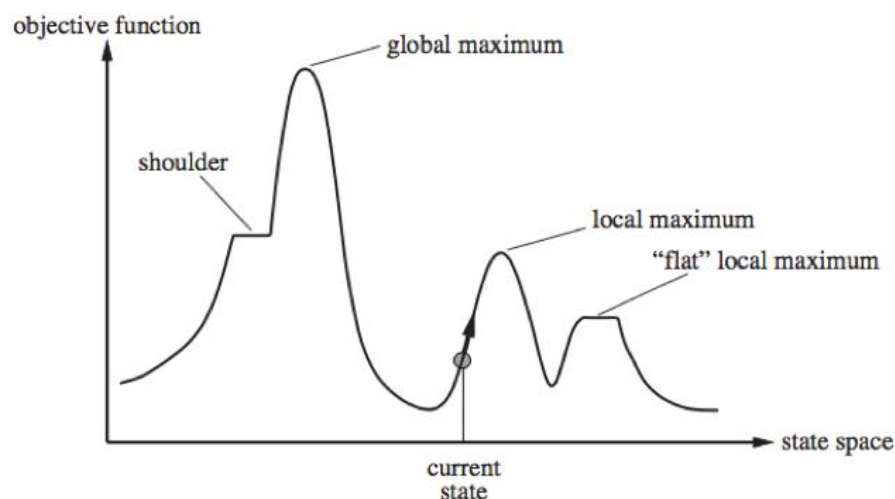
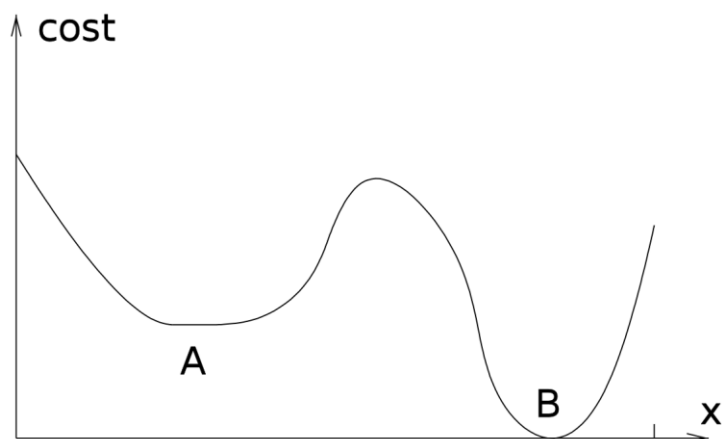
局部搜索算法

- **局部搜索算法：** 局部搜索算法从单独的一个当前状态出发，通常只移动到与之相邻的状态。
 - ◆ 典型情况下，搜索的路径是不保留的
 - ◆ 优点： 只占用少量的内存，通常可以在很大状态空间中找到合理的解
- 除了找到目标，局部搜索算法对于解决纯粹的最优化问题是很有用的，其目标是根据一个目标函数找到最佳状态。
 - ◆ 许多最优化问题不适合于“标准的”搜索模型。如自然界提供了一个目标函数——繁殖适应性——达尔文的进化论可以被视为优化的尝试，但是这个问题没有“目标测试”和“路径耗散”。

局部搜索算法：状态空间地形图

■ **局部搜索算法：** 类比在地形图中找到最高或最低点

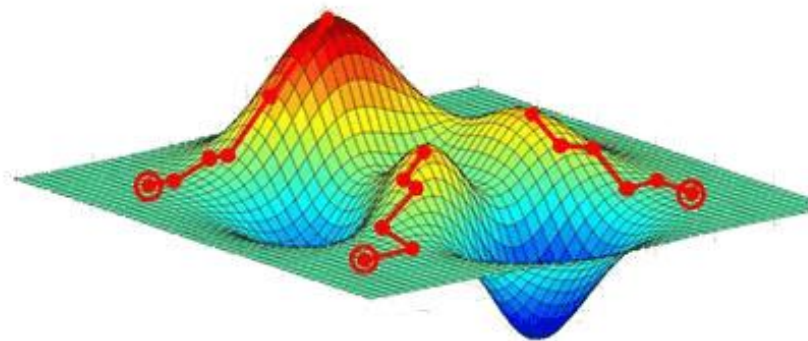
- ◆ 地形图既有“**位置**”（用**状态**定义），又有“**高度**”（由**启发式耗散函数**或**目标函数值**定义）。
- ◆ 如果高度对应于耗散，那么目标是找到最低谷——即一个**全局最小值**；
- ◆ 如果高度对应于目标函数，那么目标是找到最高峰——即一个**全局最大值**。
- ◆ 局部搜索算法就象对地形图的探索，如果存在解，那么**完备的局部搜索算法**总能找到解；**最优的局部搜索算法**总能找到全局最小值 / 最大值。



爬山法搜索

■ **爬山搜索算法** 是一种 **局部搜索算法**，可在海拔/值不断增加的方向上连续移动以找到山峰或对该问题的最佳解决方案。当它达到一个峰值，没有邻居有更高的值时，它将终止。

- 它也被称为贪婪本地搜索，因为它只查找其良好的直接邻居状态，而没有超出此范围
- 爬山算法的节点具有状态和值两个部分
- 不需要维护和处理搜索树或图形，而是仅保留一个当前状态



■ 爬山算法是一种用于优化数学问题的技术。爬山算法的一个被广泛讨论的例子是旅行推销员问题，其中我们需要最小化推销员的行进距离。当有良好的启发式功能时，通常会使用“爬山”。

爬山法搜索：8皇后问题

局部搜索算法通常使用完全状态形式化，即每个状态都在棋盘上放8个皇后，每列一个。

目标：任何一个皇后都不会攻击到其他的皇后（皇后可以攻击和它在同一行、同一列或同一对角线上的皇后）。

后继函数：返回的是移动一个皇后到和它同一列的另一个方格中的所有可能的状态。因此，每个状态有 $56=8*7$ 个后继。

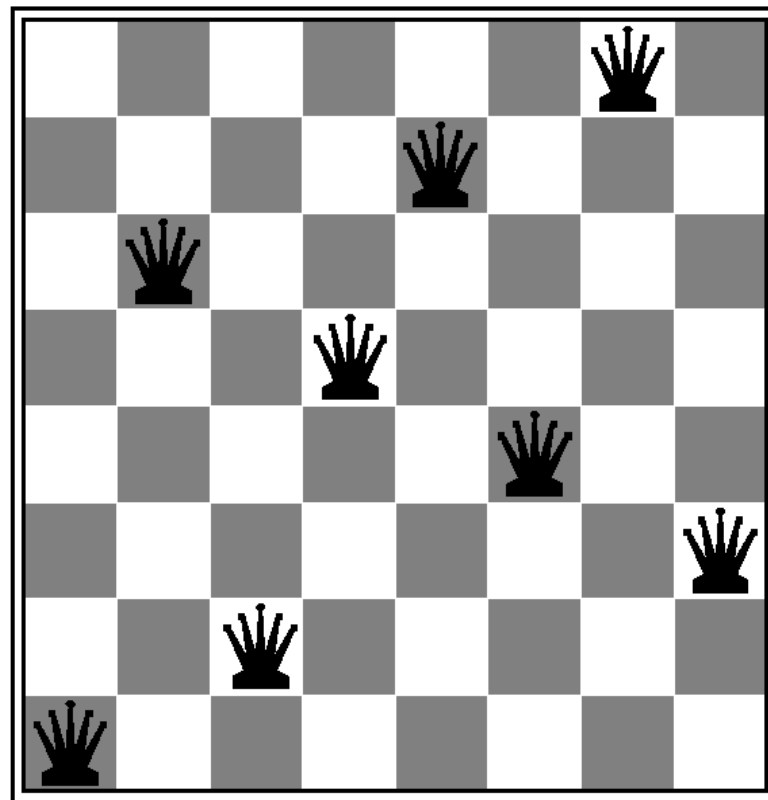
启发式耗散函数h：是可以彼此攻击的皇后对的数目，不管中间是否有障碍。
全局最小值为0，即没有彼此攻击的皇后对。

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	Q	13	16	13	16
Q	14	17	15	Q	14	16	16
17	Q	16	18	15	Q	15	Q
18	14	Q	15	15	14	Q	16
14	14	13	17	12	14	12	18

爬山法搜索：8皇后问题

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♙	13	16	13	16
♙	14	17	15	♙	14	16	16
17	♙	16	18	15	♙	15	♙
18	14	♙	15	15	14	♙	16
14	14	13	17	12	14	12	18

$h=17$ 的一个状态，
最好的后继的 $h=12$ 。



h 取局部极小值时的一个状态

爬山法搜索：

- 爬山法(hill-climbing)：是一个向值增加的方向持续移动的循环过程，即登高过程。

； 最陡上升方式的爬山搜索算法

； 如果相邻状态中没有比它更高的值，则算法结束于“峰顶”

function Hill-Climbing(*problem*)

inputs: *problem*, a problem

returns a state that is a local maximum

1. *current* \leftarrow Make-Node(Initial-State[*problem*])

2. **loop do**

3. *neighbor* \leftarrow a highest-valued successor of *current*

4. **if** Value[*neighbor*] \leq Value[*current*] **then**

return State[*current*]

5. *current* \leftarrow *neighbor*

爬山法搜索：8皇后问题

图4.1 (a)

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	Q	13	16	13	16
Q	14	17	15	Q	14	16	16
17	Q	16	18	15	Q	15	Q
18	14	Q	15	15	14	Q	16
14	14	13	17	12	14	12	18

(b)

						Q	
				Q			
	Q						
			Q				
					Q		
							Q
		Q					
Q							

爬山法能很快朝着解的方向进展。例如，从图4-1 (a) 中的状态，它只需要五步就能到达图4-1 (b) 中的状态，它的 $h=1$ ，这基本上很接近于解了。可是，爬山法经常会遇到下面的问题：

爬山法搜索：存在问题

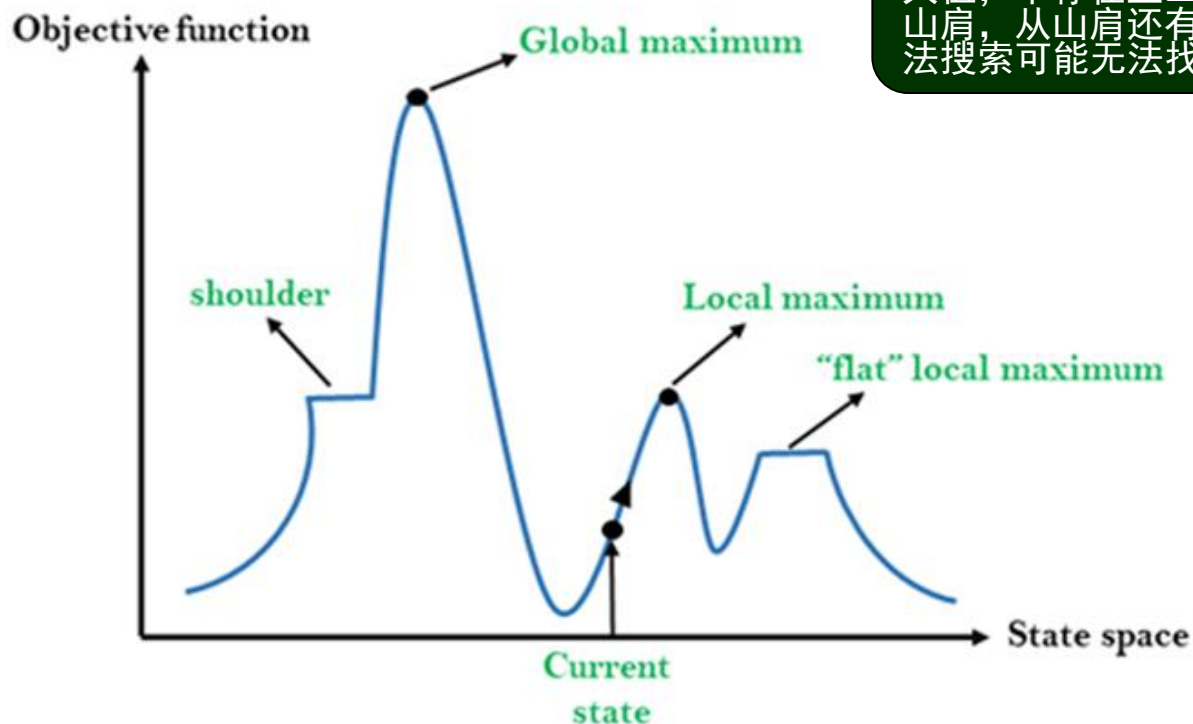
爬山法是一种贪婪局部搜索，不是最优解算法(或是不完备的)，爬山法经常会遇到下面的问题：

局部极大值：比其邻居状态都高的顶峰，但是小于全局最大值。

高原或山肩：评价函数平坦的一块区域。山肩有上坡边缘

山脊：一系列的局部极大值。

对于状态空间地形图上评价函数值平坦的一块区域。它可能是一块平的局部极大值，不存在上山的出路，或者是一个山肩，从山肩还有可能取得进展。爬山法搜索可能无法找到离开高原的道路。



➤局部极大值是一个比它的每个邻居状态都高的峰顶，但是比全局最大值要低。

➤爬山法算法到达局部极大值附近就会被拉向峰顶，然后被卡在局部极大值处无处可走。

➤更具体地，图4-1 (b) 中的状态事实上是一个局部极大值（即耗散h的局部极小值）；不管移动哪个皇后得到的情况都会比原来差。

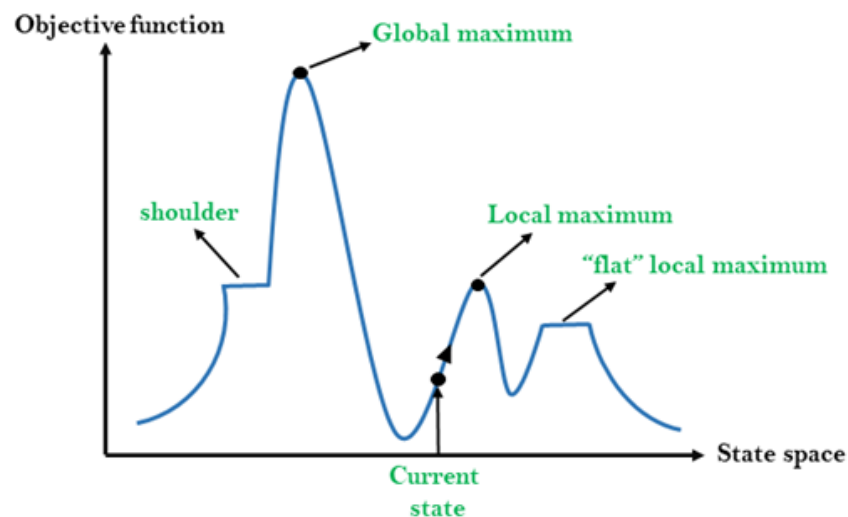
爬山法搜索：存在问题

- 对于最陡上升的爬山法算法，从一个随机生成的八皇后问题的状态开始；
 - ◆ 在86%的情况下会被卡住；
 - ◆ 只有14%的问题实例能求到最优解。
- 但这个算法速度很快，成功得到最优解的平均步数是4步，被卡住的平均步数是3步。
 - ◆ 对于包含 $8^8 \approx 1$ 千7百万个状态的状态空间是不错的结果。

爬山法搜索：解决思路

- 前述算法中，如果到达一个高原，最佳后继的状态值和当前状态值相等时将会停止。
- 如果高原其实是山肩，继续前进——即侧向移动通常是一种好方法
- 如果高原是平坦的局部最大值而不是山肩，算法会陷入死循环
 - 限制侧向移动次数。

- 例、八皇后问题中，允许最多连续侧向移动100次，将使问题实例的解决率从14%上升到94%。
- 代价是每个成功搜索实例的平均步长大约为21步，每个失败搜索的平均步长大约为64步。



爬山法搜索的变形

■ 针对爬山法的不足，有许多变化的形式

■ **随机爬山法**，它在上山移动中随机地选择下一步；选择的概率随着上山移动的陡峭程度而变化。

□ 这种算法通常比最陡上升算法的收敛速度慢不少，但是在某些状态空间地形图上能找到更好的解。

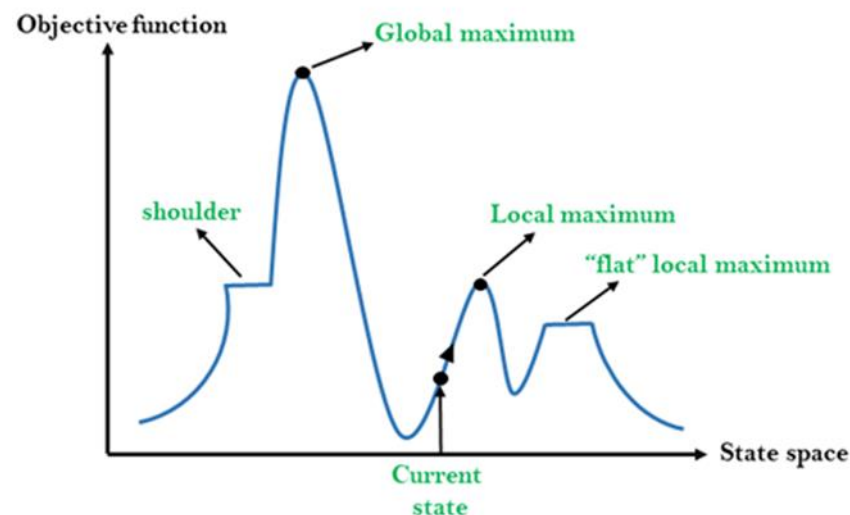
■ **首选爬山法**，它在实现随机爬山法的基础上，采用的方式是随机地生成后继节点直到生成一个优于当前节点的后继。

□ 这个算法在有很多后继节点的情况下有很好的效果。

随机重新开始爬山法：随机生成初始状态，进行一系列爬山法搜索。这时算法是完备的概率接近1。

□ 如果爬山法每次成功的概率为 p ，则需要重新开始搜索的期望次数是 $1/p$ 。

□ 对于**皇后问题**，该方法通常用不了1分钟就可以找到解。

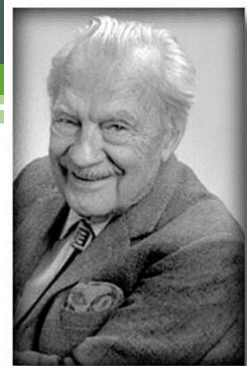


爬山法搜索的特点

- 爬山法搜索成功与否在很大程度上取决于状态空间地形图的形状。
 - 如果在图中几乎没有局部极大值和高原，随机重新开始的爬山法将会很快地找到好的解。
 - 许多实际问题的地形图存在着大量的局部极值。**NP**难题通常有指数级数量的局部极大值。
- 经过少数随机重新开始的搜索之后还是能找到一个合理的较好的局部极大值的。

高级搜索

- 爬山算法
- 模拟退火算法
- 遗传算法
- 应用案例



Nick Metropolis

模拟退火算法

- 模拟退火算法概述

- 模拟退火算法 (Simulated Annealing, SA) 是一种模拟物理退火的过程而设计的优化算法。它的基本思想最早在1953年就被Metropolis提出，但直到1983年Kirkpatrick等人才设计出真正意义上的模拟退火算法并进行应用。

- 模拟退火算法目的

- 解决NP复杂性问题；
- 克服优化过程陷入局部极小；
- 克服初值依赖性。

模拟退火算法

- 物理退火过程

- 退火是指将固体加热到足够高的温度，使分子呈随机排列状态，然后逐步降温使之冷却，最后分子以低能状态排列，固体达到某种稳定状态。

■ 加温过程——增强粒子的热运动，消除系统原先可能存在的非均匀态；

■ 等温过程——对于与环境换热而温度不变的封闭系统，系统状态的自发变化总是朝自由能减少的方向进行，当自由能达到最小时，系统达到平衡态；

■ 冷却过程——使粒子热运动减弱并渐趋有序，系统能量逐渐下降，从而得到低能的晶体结构。

$$P\{\bar{E} = E(r)\} = \frac{1}{Z(T)} \exp\left(-\frac{E(r)}{k_B T}\right)$$

\bar{E} 表示分子能量的一个随机变量， $E(r)$ 表示状态 r 的能量， $k_B > 0$ 为Boltzmann常数。 $Z(T)$ 为概率分布的标准化因子：

$$Z(T) = \sum_{s \in D} \exp\left(-\frac{E(s)}{k_B T}\right)$$

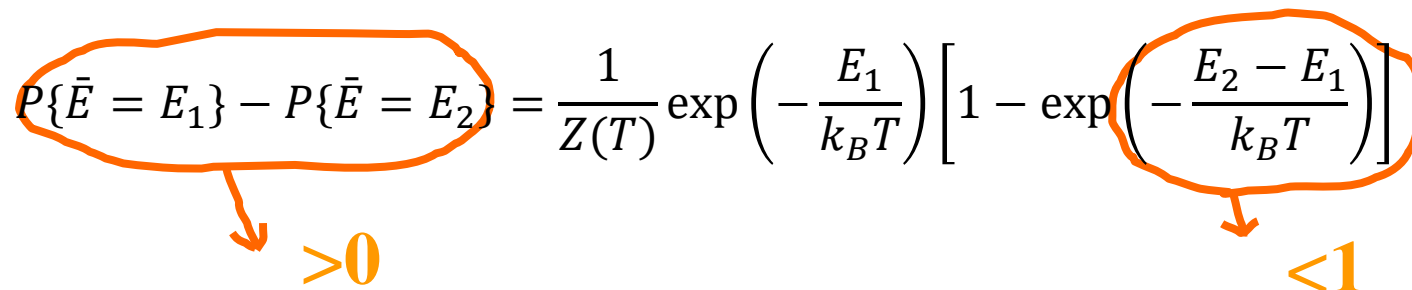
在温度 T ，分子停留在状态 r 满足Boltzmann概率分布

模拟退火算法

- 物理退火过程数学表述

- 在同一个温度 T ，选定两个能量 $E_1 < E_2$ ，有

$$P\{\bar{E} = E_1\} - P\{\bar{E} = E_2\} = \frac{1}{Z(T)} \exp\left(-\frac{E_1}{k_B T}\right) \left[1 - \exp\left(-\frac{E_2 - E_1}{k_B T}\right)\right]$$



>0 <1

在同一个温度，分子停留在能量小的状态的概率比停留在能量大的状态的概率要大。

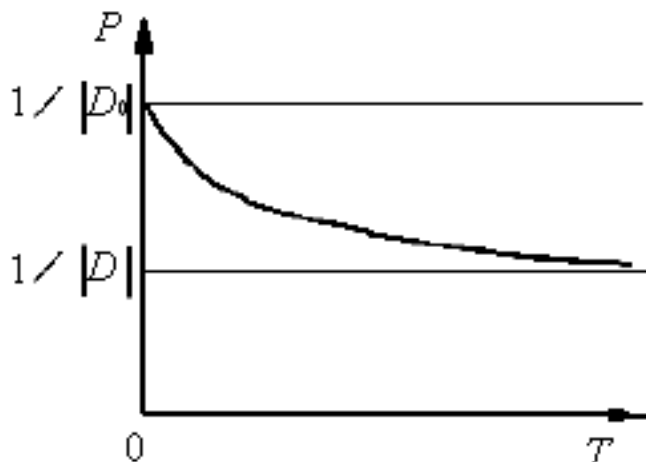
模拟退火算法

物理退火过程数学表述

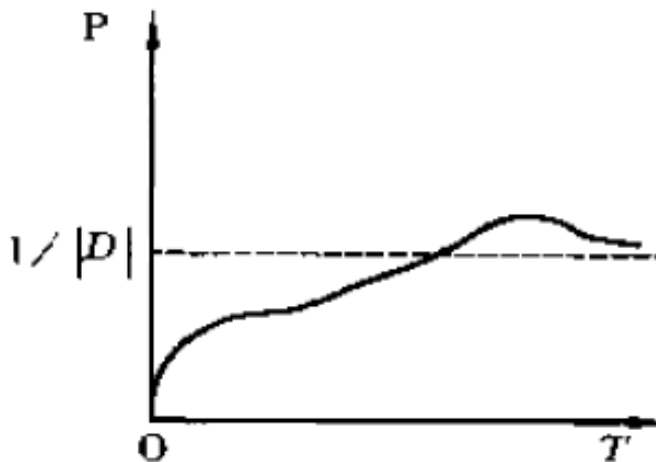
$$P\{\bar{E} = E(r)\} = \frac{1}{Z(T)} \exp\left(-\frac{E(r)}{k_B T}\right)$$

\bar{E} 表示分子能量的一个随机变量， $E(r)$ 表示状态 r 的能量， $k_B > 0$ 为Boltzmann常数。 $Z(T)$ 为概率分布的标准化因子：

$$Z(T) = \sum_{s \in D} \exp\left(-\frac{E(s)}{k_B T}\right)$$



最低能量的状态



非最低能量的状态

- 若 $|D|$ 为状态空间 D 中状态的个数， D_0 是具有最低能量的状态集合：
- 当温度很高时，每个状态概率基本相同，接近平均值 $1/|D|$ ；
- 状态空间存在超过两个不同能量时，具有最低能量状态的概率超出平均值 $1/|D|$ ；
- 当温度趋于0时，分子停留在最低能量状态的概率趋于1。

模拟退火算法

- 物理退火过程数学表述:

- ◆ **Metropolis准则（1953）——以概率接受新状态**

固体在恒定温度下达到热平衡的过程可以用**Monte Carlo方法**（计算机随机模拟方法）加以模拟，虽然该方法简单，但必须大量采样才能得到比较精确的结果，计算量很大。

模拟退火算法

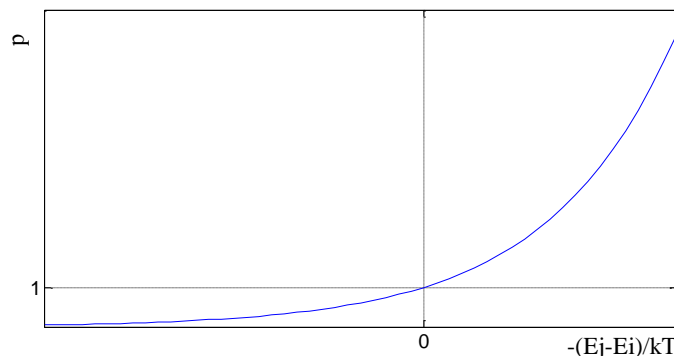
- 物理退火过程数学表述:

- Metropolis准则（1953）——以概率接受新状态

若在温度 T ，当前状态 $i \rightarrow$ 新状态 j

若 $E_j < E_i$ ，则接受 j 为当前状态；

否则，若概率 $p = \exp[-(E_j - E_i)/k_B T]$ 大于 $[0,1)$ 区间的随机数，则仍接受状态 j 为当前状态；若不成立则保留状态 i 为当前状态。



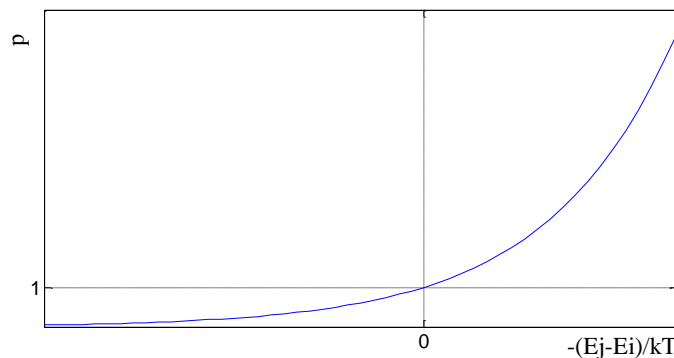
模拟退火算法

- 物理退火过程数学表述:

- Metropolis准则（1953）——以概率接受新状态

$$p = \exp[-(E_j - E_i)/k_B T]$$

在高温下，可接受与当前状态能量差较大的新状态；
在低温下，只接受与当前状态能量差较小的新状态。



模拟退火算法

- 物理退火过程
 - 退火是指将固体加热到足够高的温度，使分子呈随机排列状态，然后逐步降温使之冷却，最后分子以低能状态排列，固体达到某种稳定状态。
- 模拟退火算法思想
 - 模拟退火算法采用类似于物理退火的过程，先在一个高温状态下（相当于算法随机搜索,大概率接受劣解），然后逐渐退火，徐徐冷却（接受劣解概率变小直至为零，相当于算法局部搜索），最终达到物理基态（相当于算法找到最优解）。算法的本质是通过温度来控制算法接受劣解的概率（劣向转移是脱出局部极小的核心机制）。

模拟退火算法思想

物理退火过程

物体内部的状态

状态的能量

温度

溶解过程

退火冷却过程

状态的转移

能量最低状态

类比关系

模拟退火算法

问题的解空间

解的质量

控制参数

设定初始温度

控制参数的修改

解在邻域中的变化

最优解

模拟退火基本流程

◆ 基本步骤

给定初温 $t=t_0$, 随机产生初始状态 $s=s_0$, 令 $k=0$;

Repeat

Repeat

产生新状态 $s_j=\text{Genete}(s)$;

if $\min\{1, \exp[-(C(s_j)-C(s))/t_k]\} \geq \text{randrom}[0,1]$ $s=s_j$;

Until 抽样稳定准则满足;

退温 $t_{k+1}=\text{update}(t_k)$ 并令 $k=k+1$;

Until 算法终止准则满足;

输出算法搜索结果。

模拟退火基本流程

- ◆ 影响优化结果的主要因素

给定初温 $t=t_0$, 随机产生初始状态 $s=s_0$, 令 $k=0$;

Repeat

Repeat

产生新状态 $s_j = \text{Genete}(s)$;

if $\min\{1, \exp[-(C(s_j) - C(s))/t_k]\} \geq \text{randrom}[0,1]$ $s = s_j$;

Until 抽样稳定准则满足;

退温 $t_{k+1} = \text{update}(t_k)$ 并令 $k = k + 1$;

Until 算法终止准则满足;

输出算法搜索结果。

三函数两准则

初始温度

模拟退火基本要素

- 从算法流程上看，模拟退火（SA）算法包括三函数两准则，即状态产生函数、状态接受函数、温度更新函数、内循环终止准则和外循环终止准则，这些环节的设计将决定SA算法的优化性能。此外，初温的选择对SA算法性能也有很大影响。
- 理论上，SA算法的参数只有满足算法的收敛条件，才能保证实现的算法依概率1收敛到全局最优解。然而，SA算法的某些收敛条件无法严格实现，即使某些收敛条件可以实现，但也常常会因为实际应用的效果不理想而不被采用。因此，至今SA算法的参数选择依然是一个难题，通常只能依据一定的启发式准则或大量的实验加以选取。

模拟退火基本要素

• 状态产生函数

- 设计状态产生函数（邻域函数）的出发点应该是尽可能保证**产生的候选解遍布全部解空间**。通常，状态产生函数由两部分组成，即产生候选解的方式和候选解产生的概率分布。
- 前者决定由当前解产生候选解的方式，后者决定在当前解产生的候选解中选择不同状态的概率。
- 候选解的产生方式由问题的性质决定，通常**在当前状态的邻域结构内以一定概率方式产生**，而邻域函数和概率方式可以多样化设计，其中概率分布可以是均匀分布、正态分布、指数分布、柯西分布等。在组合优化中，通常采用解的局部搜索操作，如0-1背包问题用翻位操作，TSP问题中用交换两个城市，翻转城市片段等操作。

模拟退火基本要素

• 状态接受函数

- 状态接受函数一般以概率的方式给出，不同接受函数的差别主要在于接受概率的形式不同。设计状态接受概率，应该遵循以下原则：
 - 在固定温度下，接受使目标函数值下降的候选解的概率要大于使目标函数值上升的候选解的概率；
 - 随温度的下降，接受使目标函数值上升的解的概率要逐渐减小；
 - 当温度趋于零时，只能接受目标函数值下降的解。
- 状态接受函数的引入是SA 算法实现全局搜索的最关键的因素，但实验表明，状态接受函数的具体形式对算法性能的影响不显著。因此，SA 算法中通常采用 $\min[1, \exp(-\Delta C/t)]$ 作为状态接受函数。

模拟退火基本要素

• 初温

- 初始温度 t 、温度更新函数、内循环终止准则和外循环终止准则通常被称为退火历程(annealing schedule)。
- 实验表明，初温越大，获得高质量解的几率越大，但花费的计算时间将增加。因此，初温的确定应折衷考虑优化质量和优化效率，常用方法包括：
 - 均匀抽样一组状态，以各状态目标值的方差为初温；
 - 随机产生一组状态，确定两两状态间的最大目标值差 $|\Delta \max|$ ，然后依据差值，利用一定的函数确定初温。譬如 $t_0 = -|\Delta \max| / \ln Pr$ ，其中 Pr 为初始接受概率。若取 Pr 接近1，且初始随机产生的状态能够一定程度上表征整个状态空间时，算法将以几乎等同的概率接受任意状态，完全不受极小解的限制；
 - 利用经验公式给出。

模拟退火基本要素

• 温度更新函数

- 温度更新函数，即温度的下降方式，用于在外循环中修改温度值。
- 在非时齐SA 算法收敛性理论中，更新函数可采用指数函数。

◆ 如时齐算法的温度下降函数

(1) $t_{k+1} = \alpha t_k, k \geq 0, 0 < \alpha < 1$, α 越接近1温度下降越慢，且其大小可以不断变化；

(2) $t_k = \frac{K-k}{K} t_0$, 其中 t_0 为起始温度， K 为算法温度下降的总次数

若固定每一温度，算法均计算至平稳分布，然后下降温度，则称为**时齐算法**；
若无需各温度下算法均达到平稳分布，但温度需按一定速率下降，则称为**非时齐算法**。

模拟退火基本要素

• 内循环终止准则

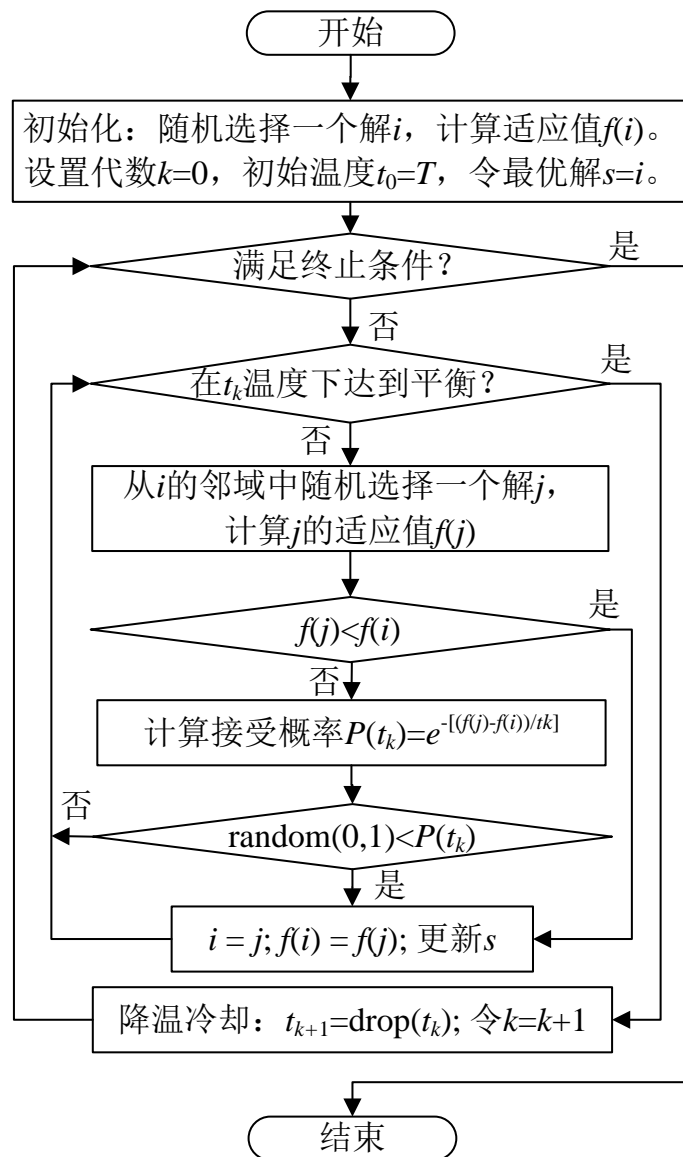
- 内循环终止准则，或称Metropolis 抽样稳定准则，用于决定在各温度下产生候选解的数目。
- 在非时齐SA算法理论中，由于在每个温度下只产生一个或少量候选解，所以不存在选择内循环终止准则的问题。
- 在时齐SA算法理论中，收敛性条件要求在每个温度下产生候选解数目趋于无穷大，以使相应的马氏链达到平稳概率分布，显然在实际应用算法时这是无法实现的，常用的抽样稳定准则包括：
 - 检验目标函数的均值是否稳定；
 - 连续若干步的目标值变化较小；
 - 按一定的步数抽样。

模拟退火基本要素

• 外循环终止准则

- 外循环终止准则，即算法终止准则，用于决定算法何时结束。设置温度终值 t_e 是一种简单的方法。SA 算法的收敛性理论中要求 t_e 趋于零，这显然是不实际的。通常的做法包括：
 - 设置终止温度的阈值；
 - 设置外循环迭代次数（内循环*外循环=算法总循环，给定总循环数，合理分配内外循环，如 $10*1000=10000$ ）；
 - 算法搜索到的最优值连续若干步保持不变；
 - 检验系统熵是否稳定。

模拟退火基本流程



//功能: 模拟退火算法伪代码

//说明: 本例以求问题最小值为目标

//参数: T 为初始温度; L 为内层循环次数

procedure SA

//Initialization

Randomly generate a solution X_0 , and calculate its fitness value $f(X_0)$;

$X_{best} = X_0$; $k = 0$; $t_k = T$;

while not stop

//The search loop under the temperature t_k

for $i = 1$ to L //The loop times

Generate a new solution X_{new} based on the current solution X_k , and calculate its fitness value $f(X_{new})$.

if $f(X_{new}) < f(X_k)$

$X_k = X_{new}$;

if $f(X_k) < f(X_{best})$ $X_{best} = X_k$;

continues;

end if

Calculate $P(t_k) = e^{-[(f(X_{new}) - f(X_k)) / t_k]}$;

if $\text{random}(0,1) < P$

$X_k = X_{new}$;

end if

end for

//Drop down the temperature

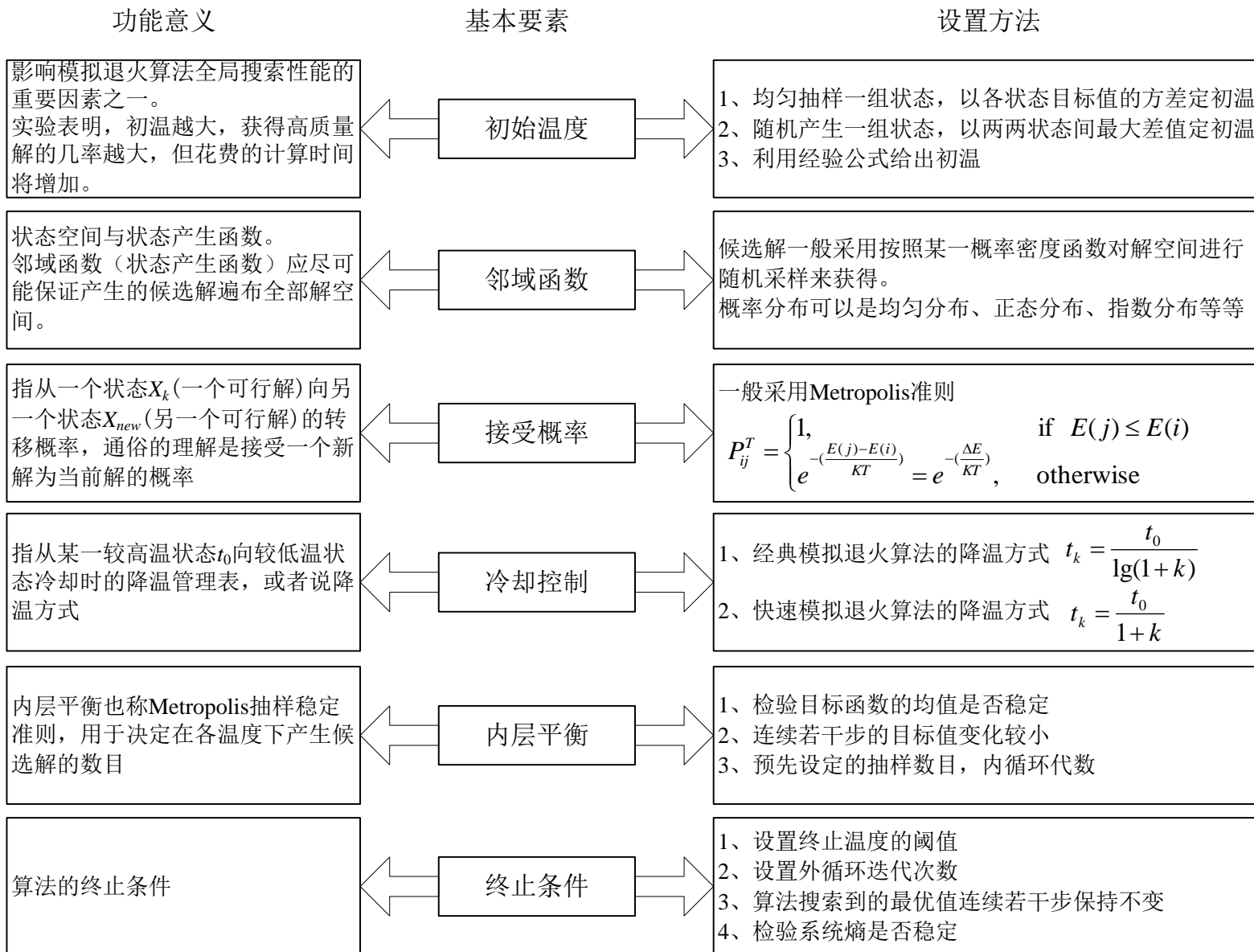
$t_{k+1} = \text{drop}(t_k)$; $k = k + 1$;

end while

print X_{best}

end procedure

模拟退火基本要素与设置



模拟退火算法的优缺点

- ◆ 模拟退火算法的优点

质量高;

初值鲁棒性强;

简单、通用、易实现。

- ◆ 模拟退火算法的缺点

由于要求较高的初始温度、较慢的降温速率、较低的终止温度，以及各温度下足够多次的抽样，因此优化过程较长。

模拟退火应用案例

- **例题** 已知背包的装载量为 $c=8$ ，现有 $n=5$ 个物品，它们的重量和价值分别是 $(2, 3, 5, 1, 4)$ 和 $(2, 5, 8, 3, 6)$ 。试使用模拟退火算法求解该背包问题，写出关键的步骤。
- **求解**：假设问题的一个可行解用0和1的序列表示，例如 $i=(1010)$ 表示选择第1和第3个物品，而不选择第2和第4个物品。用模拟退火算法求解关键过程如图所示：

模拟退火应用案例

已知：

物体个数： $n=5$

背包容量： $c=8$

重量 $w = (2, 3, 5, 1, 4)$

价值 $v = (2, 5, 8, 3, 6)$

第一步：初始化。假设初始解为 $i=(11001)$ ，初始温度为 $T=10$ 。计算 $f(i)=2+5+6=13$ ，最优解 $s=i$

第三步：降温，假设温度降为 $T=9$ 。如果没有达到结束标准，则返回第二步继续执行

假设在继续运行的时候，从当前解 $i=(10110)$ 得到一个新解 $j=(00111)$ ，这时候的函数值为 $f(j)=8+3+6=17$ ，这是一个全局最优解。可见上面过程中接受了劣解是有好处的。

第二步：在 T 温度下局部搜索，直到“平衡”，假设平衡条件为执行了3次内层循环。

(2-1) 产生当前解 i 的一个邻域解 j （如何构造邻域根据具体的问题而定，这里假设为随机改变某一位的0/1值或者交换某两位的0/1值），假设 $j=(11100)$ 要注意产生的新解的合法性，要舍弃那些总重量超过背包装载量的非法解

(2-2) $f(j) = 2+5+8=15 > 13=f(i)$ ，所以接受新解 $i=j$ ； $f(i)=f(j)=15$ ；而且 $s=i$ ；
要注意求解的是最大值，因此适应值越大越优

(2-3) 返回 (2-1) 继续执行。

(a) 假设第二轮得到的新解 $j=(11010)$ ，由于 $f(j) = 2+5+3=10 < 15=f(i)$ ，所以需要计算接受概率 $P(T)=\exp((f(j)-f(i))/T) = \exp(-0.5) = 0.607$ ，假设 $\text{random}(0,1) > P(T)$ ，则不接受新解

(b) 假设第三轮得到的新解 $j=(10110)$ ，由于 $f(j) = 2+8+3=13 < 15=f(i)$ ，所以需要计算接受概率 $P(T)=\exp((f(j)-f(i))/T) = \exp(-0.3) = 0.741$ ，假设 $\text{random}(0,1) < P(T)$ ，则接受新解
按照一定的概率接受劣解，也是跳出局部最优的一种手段

(2-4) 这时候， T 温度下的“平衡”已达到（即已经完成了3次的邻域产生），结束内层循环

模拟退火改进策略

- 在确保一定要求的优化质量基础上，提高模拟退火算法的搜索效率（时间性能），是对SA 算法进行改进的主要内容。可行的方案包括：
 - 设计合适的状态产生函数，使其根据搜索进程的需要表现出状态的全空间分散性或局部区域性；
 - 设计高效的退火历程；
 - 避免状态的迂回搜索；
 - 采用并行搜索结构；
 - 为避免陷入局部极小，改进对温度的控制方式；
 - 选择合适的初始状态；
 - 设计合适的算法终止准则。

高级搜索

- 爬山算法
- 模拟退火算法
- 遗传算法
- 应用案例

遗传算法：进化算法

- **进化算法** (evolutionary algorithms, EA) 是基于自然选择和自然遗传等生物进化机制的一种搜索算法。
- 生物进化是通过繁殖、变异、竞争和选择实现的；而进化算法则主要通过选择、重组和变异这三种操作实现优化问题的求解。
- 进化算法是一个“算法簇”，包括遗传算法(GA)、遗传规划、进化策略和进化规划等。
- 进化算法的基本框架是遗传算法所描述的框架。

遗传算法：进化算法

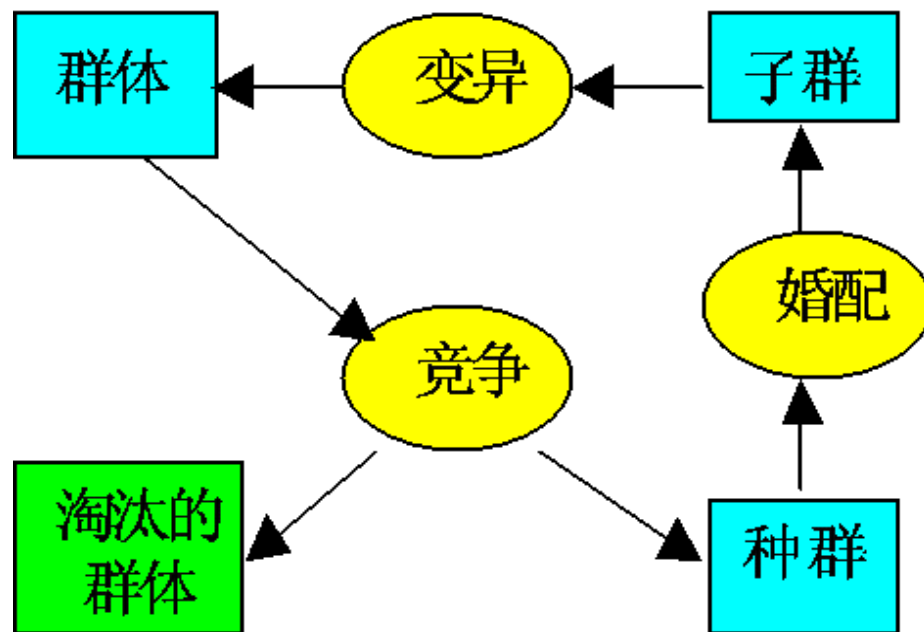
- **适者生存**：最适合自然环境的群体往往产生了更大的后代群体。
- 生物进化的基本过程：

染色体(chromosome)：生物的遗传物质的主要载体。

基因(gene)：扩展生物性状的遗传物质的功能单元和结构单位。

基因座 (locus)：染色体中基因的位置。

等位基因 (alleles)：基因所取的值。

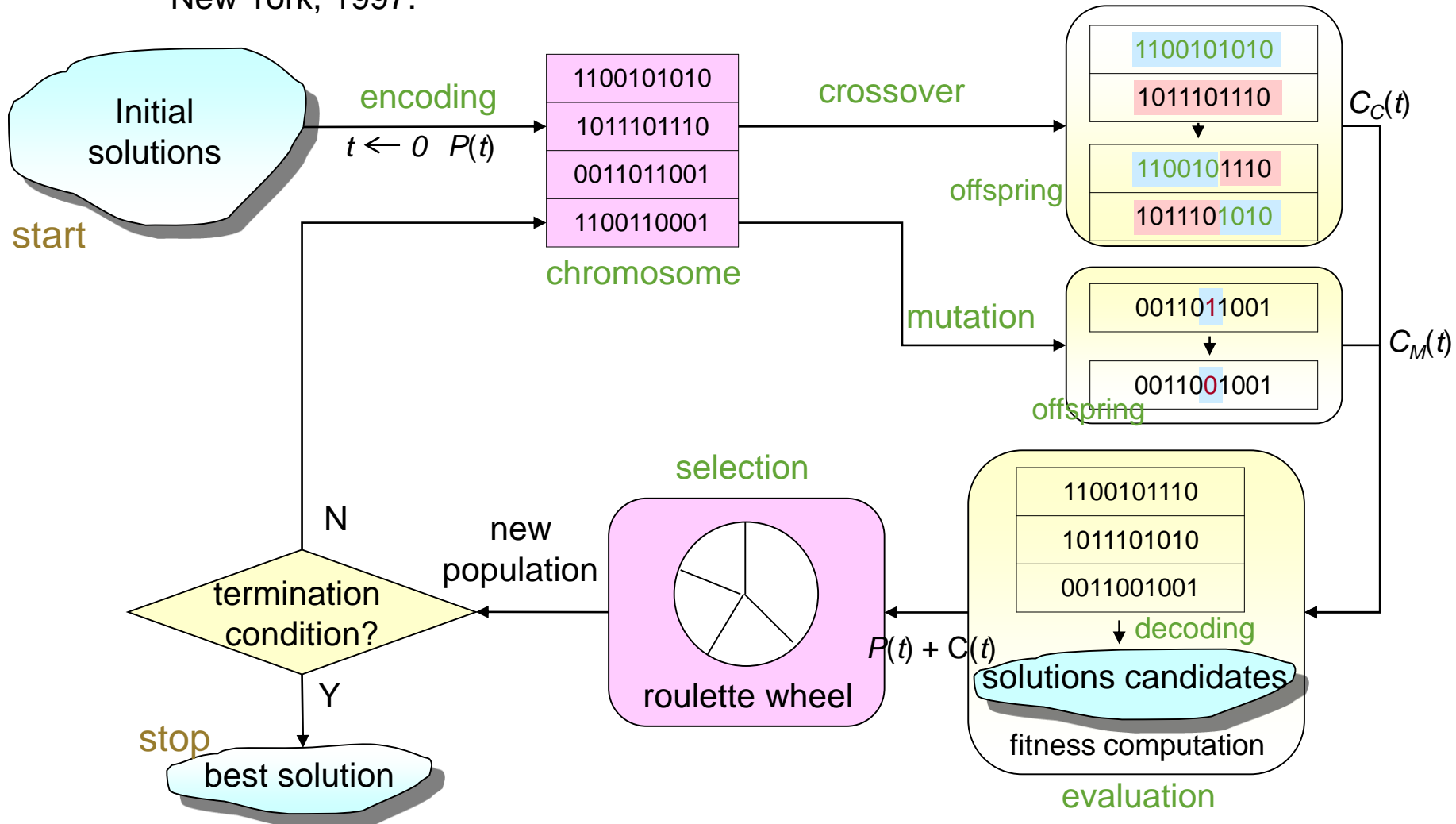


遗传算法的概念

- **遗传算法**（genetic algorithms, GA）：一类借鉴生物界自然选择和自然遗传机制的随机搜索算法，非常适用于处理传统搜索方法难以解决的复杂和非线性优化问题。
- 遗传算法可广泛应用于组合优化、机器学习、自适应控制、规划设计和人工生命等领域。

遗传算法的概念 The general structure of genetic algorithms

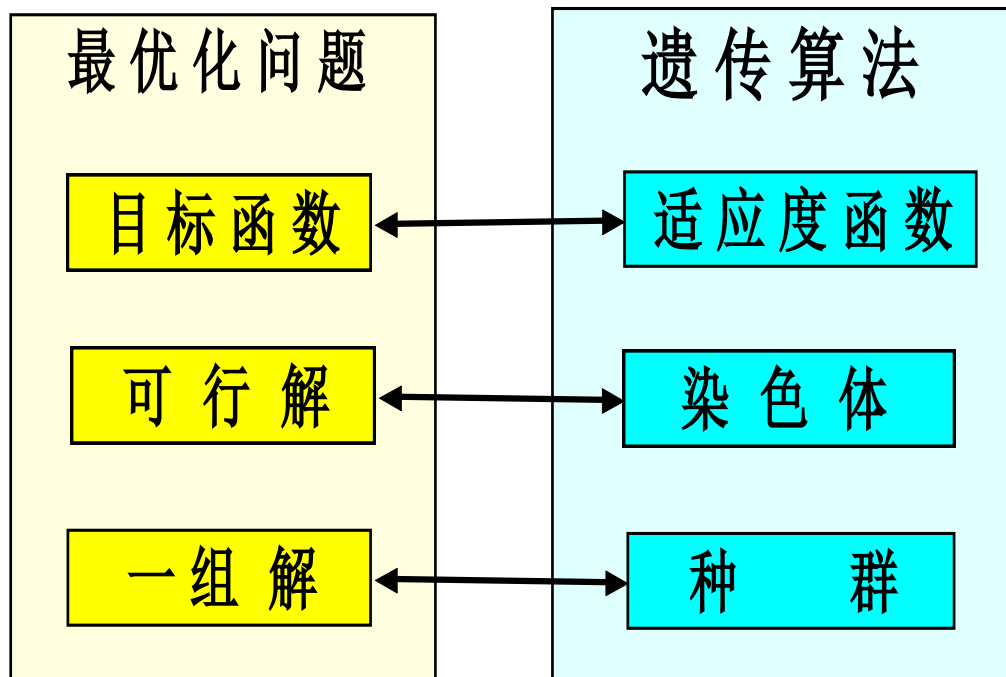
Gen, M. & R. Cheng: *Genetic Algorithms and Engineering Design*, John Wiley, New York, 1997.



遗传算法概念对比

生物遗传概念	遗产算法中的应用
适者生存	目标值比较大的解被选择的可能性大
个体（Individual）	解
染色体（Chromosome）	解的编码（字符串、向量等）
基因（Gene）	解的编码中每一分量
适应性（Fitness）	适应度函数值
群体（Population）	根据适应度值选定的一组解（解的个数为群体的规模）
婚配（Marry）	交叉（Crossover）选择两个染色体进行交叉产生一组新的染色体的过程
变异（Mutation）	编码的某一分量发生变化的过程

遗传算法概念对比



■ 遗传算法的基本思想：

在求解问题时从多个解开始，然后通过一定的法则进行逐步迭代以产生新的解。

遗传算法发展历程

- 1962年，Fraser提出了自然遗传算法。
- 1965年，Holland首次提出了人工遗传操作的重要性。
- 1967年，Bagley首次提出了遗传算法这一术语。
- 1970年，Cavicchio把遗传算法应用于模式识别中。
- 1971年，Hollstien在论文《计算机控制系统中人工遗传自适应方法》中阐述了遗传算法用于数字反馈控制的方法。
- 1975年，美国J. Holland出版了《自然系统和人工系统的适配》；DeJong完成了重要论文《遗传自适应系统的行为分析》。
- 20世纪80年代以后，遗传算法进入兴盛发展时期。

遗传算法步骤 Procedure of Simple GA

procedure: Simple GA

input: GA parameters

output: best solution

begin

$t \leftarrow 0$; // t : generation number

initialize $P(t)$ by encoding routine 编码; // $P(t)$: population of chromosomes

fitness $eval(P)$ by decoding routine 解码; 初代种群产生

while (not termination condition) do

crossover 交叉 $P(t)$ to yield $C(t)$; // $C(t)$: offspring

mutation 变异 $P(t)$ to yield $C(t)$;

fitness 适应度计算 $eval(C)$ by decoding routine 解码;

select 选择子代 $P(t+1)$ from $P(t)$ and $C(t)$;

$t \leftarrow t+1$;

end

output best solution;

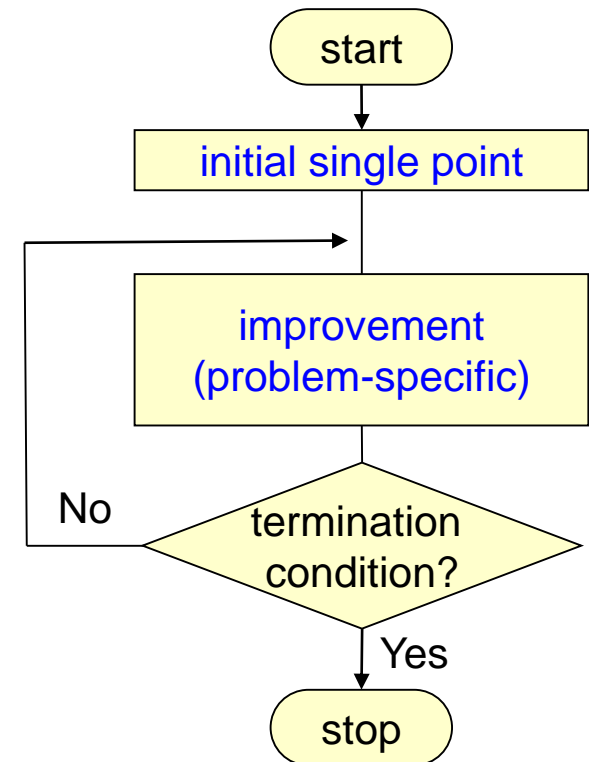
end

遗传算法与其他搜索算法对比 Major Advantages

□ Conventional Method (point-to-point approach)

- Generally, algorithm for solving optimization problems is a **sequence of computational steps** which asymptotically converge to optimal solution.
- Most of classical optimization methods generate a deterministic sequence of **computation based on the gradient or higher order** derivatives of objective function.
- The methods are applied to a single point in the search space.
- The point is then improved along the **deepest descending** direction gradually through iterations.
- This **point-to-point approach** takes the danger of falling in local optima.

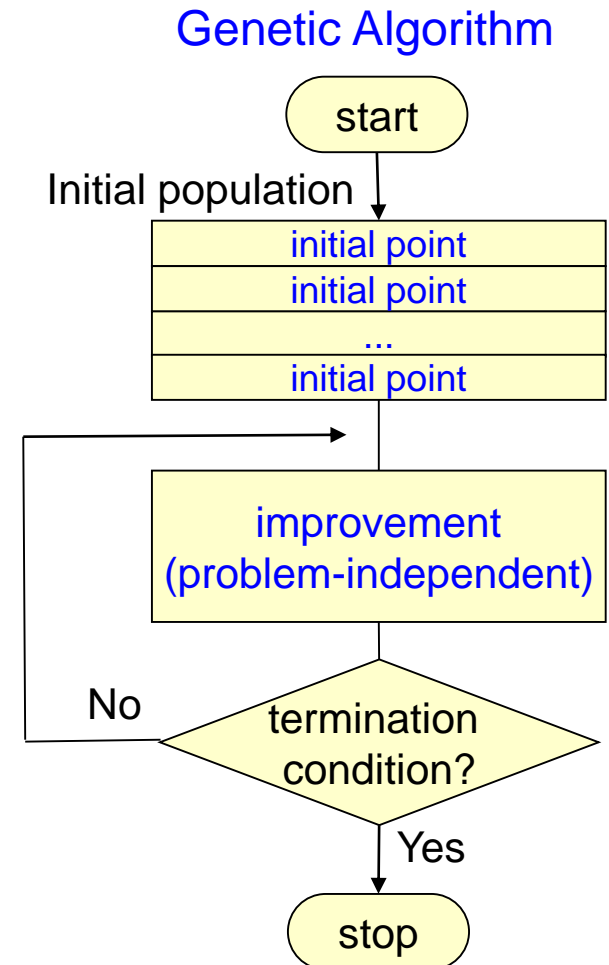
Conventional Method



遗传算法与其他搜索算法对比 Major Advantages

□ Genetic Algorithm (population-to-population approach)

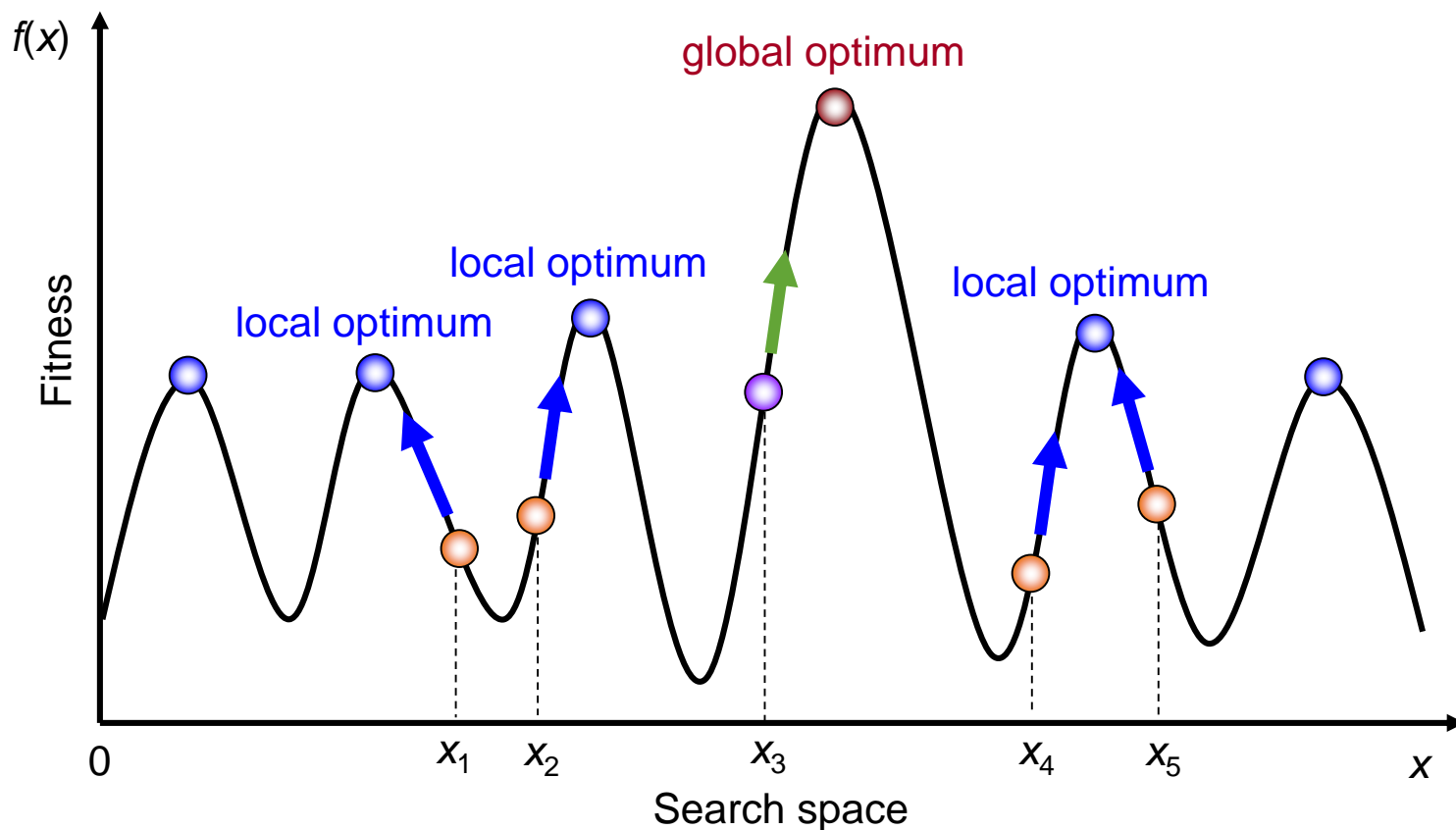
- Genetic algorithms performs a **multiple directional search** by maintaining a population of potential solutions.
- The **population-to-population approach** is hopeful to make the search escape from local optima.
- Population undergoes a simulated evolution: at each generation the relatively **good solutions are reproduced**, while the relatively bad solutions die.
- Genetic algorithms use **probabilistic transition rules** to select someone to be reproduced and someone to die so as to guide their search toward regions of the search space with likely improvement.



遗传算法与其他搜索算法对比

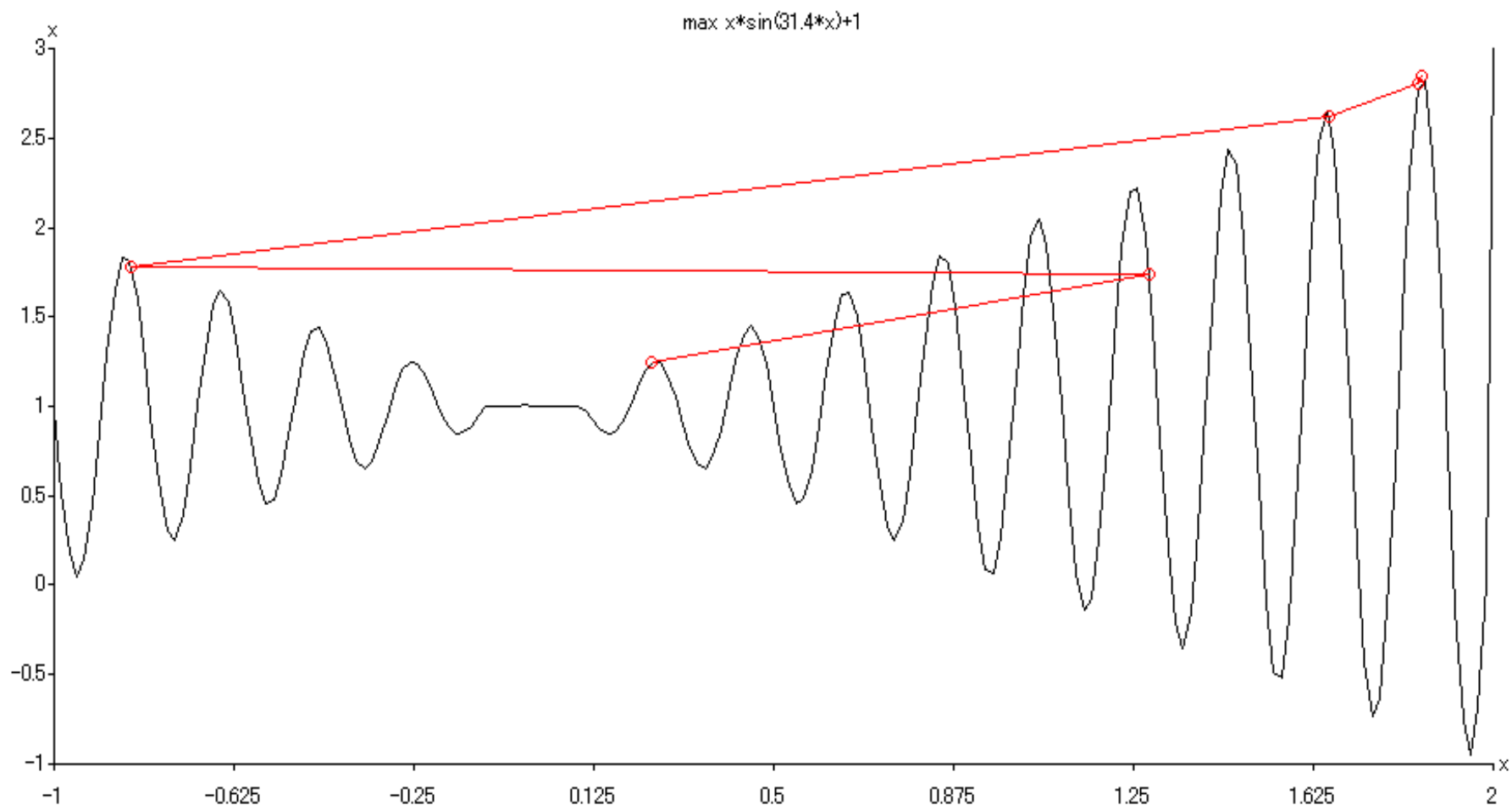
□ Random Search + Directed Search

$$\begin{array}{ll} \max & f(x) \\ \text{s. t.} & 0 \leq x \leq u_b \end{array}$$



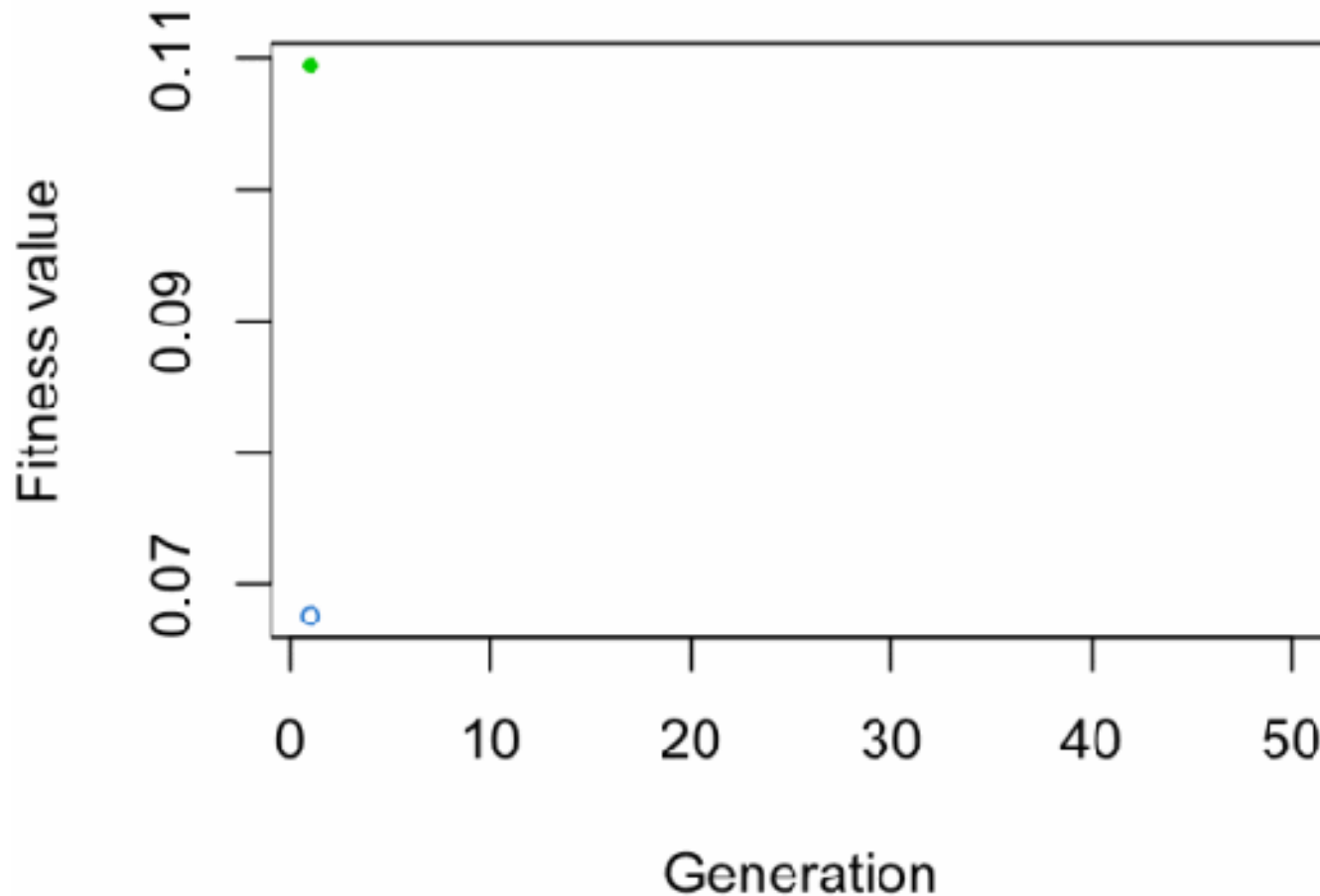
遗传算法与其他搜索算法对比

- Example of Genetic Algorithm for Unconstrained Numerical Optimization ([Michalewicz, 1996](#))



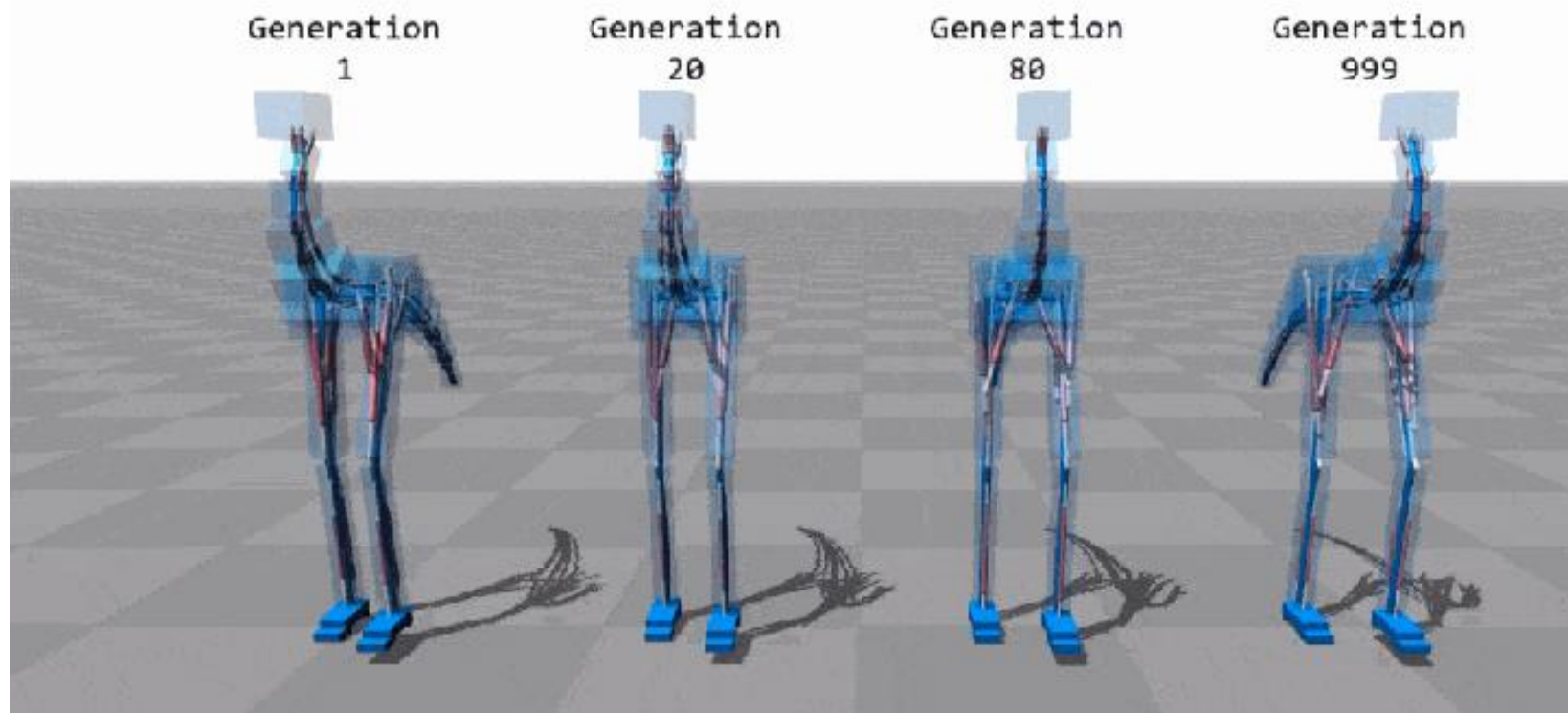
遗传算法迭代

Iteration 1



- **Blue dot:** Population fitness average
- **Green dot:** Best fitness value

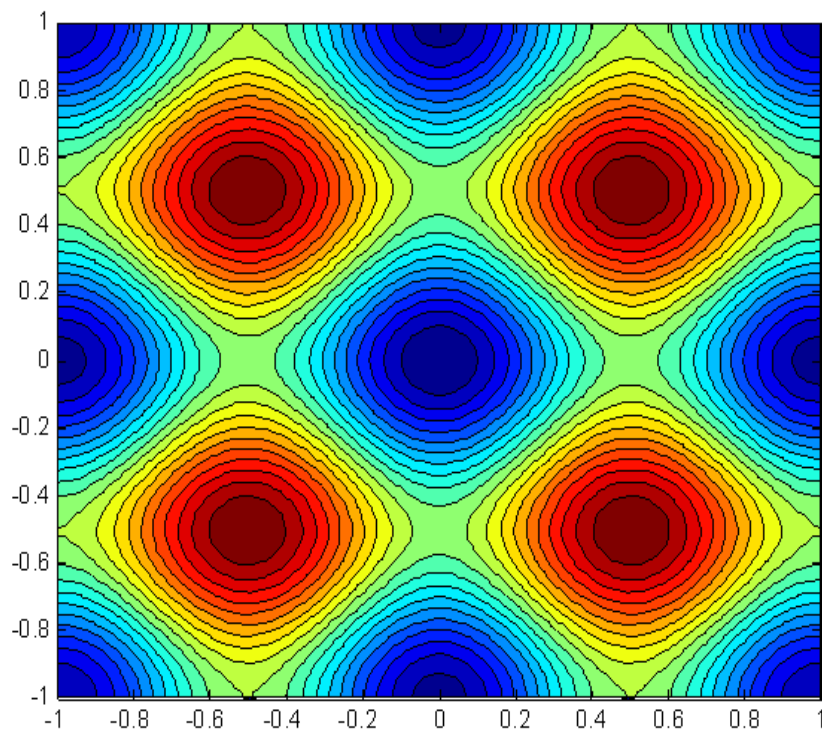
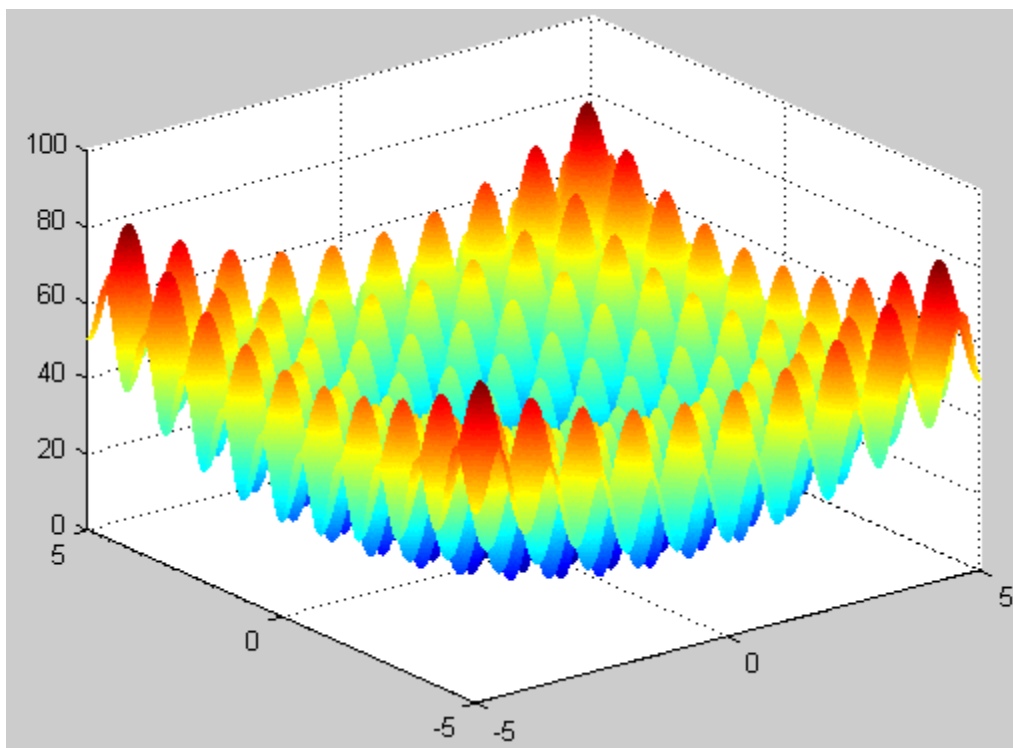
遗传算法迭代



遗传算法案例

□ 例：用遗传算法求解下面一个Rastrigin函数的最小值。

$$f(x_1, x_2) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2)$$
$$-5 \leq x_i \leq 5 \quad i = 1, 2$$



1 Example with Simple Genetic Algorithms

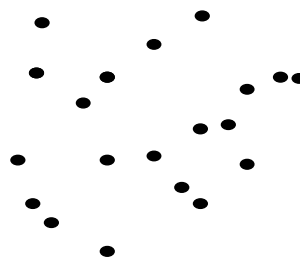
https://rednuht.org/genetic_walkers/

□ 例：用遗传算法求解下面一个Rastrigin函数的最小值。

$$f(x_1, x_2) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2)$$
$$-5 \leq x_i \leq 5 \quad i = 1, 2$$

• 初始种群：

x_1 x_2



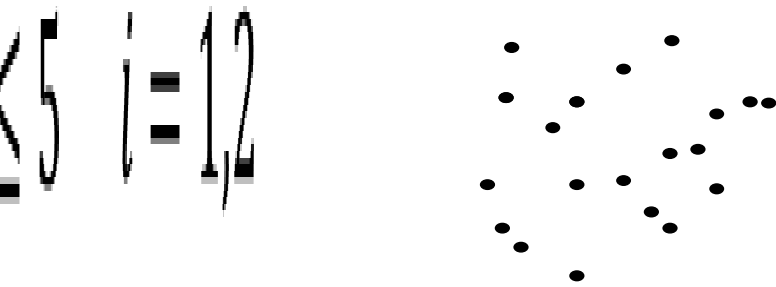
1 Example with Simple Genetic Algorithms

遗传算法案例

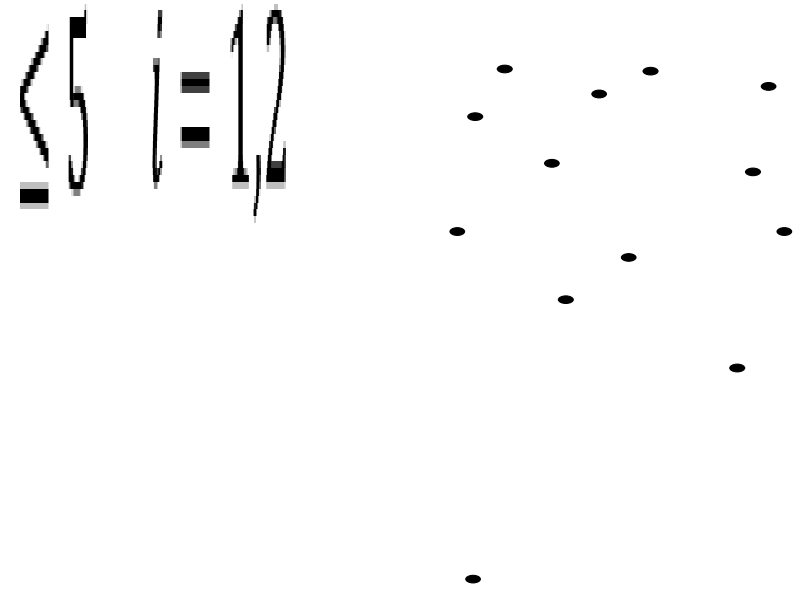
□ 例：用遗传算法求解下面一个Rastrigin函数的最小值。

$$f(x_1, x_2) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2)$$
$$-5 \leq x_i \leq 5 \quad i = 1, 2$$

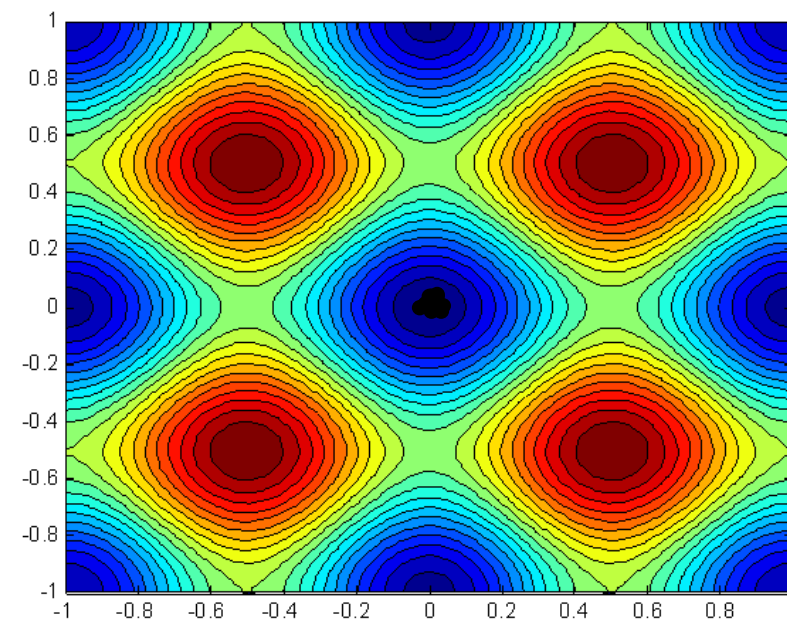
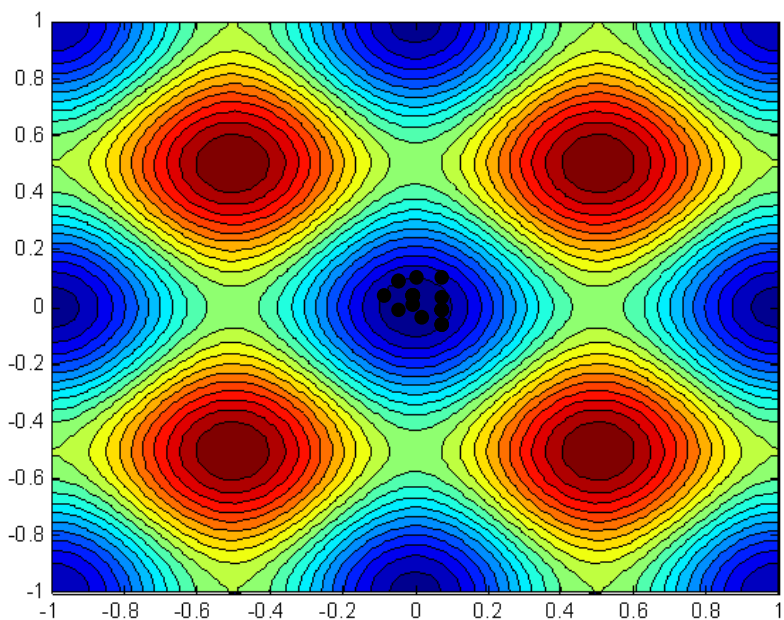
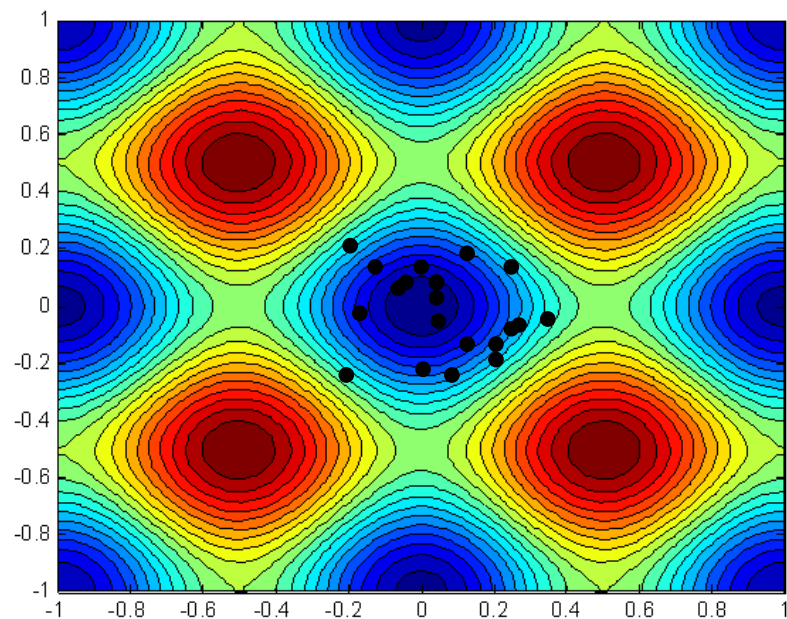
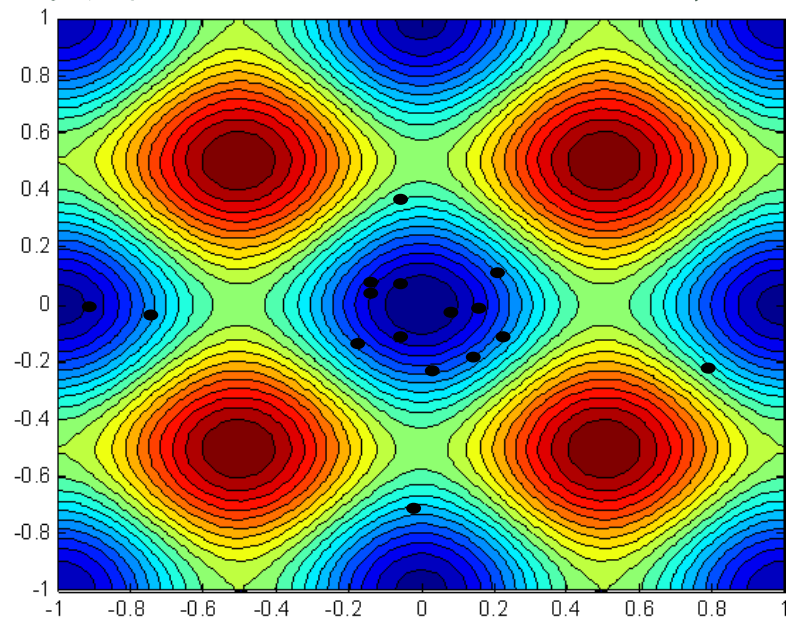
初始种群



第二代种群



• 迭代60、80、95、100次时的种群



遗传算法步骤详解

编码 初代种群 解码 (适应度) 选择 交叉 变异

procedure: Simple GA

input: GA parameters

output: best solution

begin

$t \leftarrow 0$; // t : generation number

initialize $P(t)$ by encoding routine 编码; // $P(t)$: population of chromosomes

fitness $eval(P)$ by decoding routine 解码; 初代种群产生

while (not termination condition) do

crossover 交叉 $P(t)$ to yield $C(t)$; // $C(t)$: offspring

mutation 变异 $P(t)$ to yield $C(t)$;

fitness 适应度计算 $eval(C)$ by decoding routine 解码;

select 选择子代 $P(t+1)$ from $P(t)$ and $C(t)$;

$t \leftarrow t+1$;

end

output best solution;

end

遗传算法

1. 位串编码

一维染色体编码方法：将问题状态空间的参数编码为一维排列的染色体的方法。



(1) 二进制编码

二进制编码：用若干二进制数表示一个个体，将原问题的解空间映射到位串空间 $B=\{0, 1\}$ 上，然后在位串空间上进行遗传操作。

遗传算法

(1) 二进制编码 (续)

● 优点:

类似于生物染色体的组成, 算法易于用生物遗传理论解释, 遗传操作如交叉、变异等易实现; 算法处理的模式数最多。

● 缺点:

① 相邻整数的二进制编码可能具有较大的Hamming距离, 降低了遗传算子的搜索效率。

15: 01111

16: 10000

② 要先给出求解的精度。

③ 求解高维优化问题的二进制编码串长, 算法的搜索效率低。

遗传算法

(2) Gray 编码

相邻位间转换时，只有一位产生变化，减少了由一个状态到下一个状态时逻辑的混淆

0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001

Gray编码:将二进制编码通过一个变换进行转换得到的编码。

二进制串 $\langle \beta_1 \beta_2 \dots \beta_n \rangle$

Gray $\langle \gamma_1 \gamma_2 \dots \gamma_n \rangle$

二进制编码 \rightarrow Gray编码

Gray编码 \rightarrow 二进制编码

$$\gamma_k = \begin{cases} \beta_1 & k = 1 \\ \beta_{k-1} \oplus \beta_k & k > 1 \end{cases}$$

$$\beta_k = \sum_{i=1}^k \gamma_i (\text{mod } 2)$$

遗传算法

2 实数编码

■ 采用实数表达法不必进行数制转换，可直接在解的表现型上进行遗传操作。

3 多参数映射编码

多参数映射编码的基本思想：把每个参数先进行二进制编码得到子串，再把这些子串连成一个完整的染色体。

■ 多参数映射编码中的每个子串对应各自的编码参数，所以，可以有不同的串长度和参数的取值范围。

遗传算法

1. 初始种群的产生

- 随机产生群体规模数目的个体作为初始群体。
- 随机产生一定数目的个体，从中挑选最好的个体加到初始群体中。这种过程不断迭代，直到初始群体中个体数目达到了预先确定的规模。
- 根据问题固有知识，把握最优解所占空间在整个问题空间中的分布范围，然后，在此分布范围内设定初始群体。

遗传算法

2. 种群规模的确定

■ 群体规模太小，遗传算法的优化性能不太好，易陷入局部最优解。

■ 群体规模太大，计算复杂。

■ **模式定理Schema Theorem**表明：若群体规模为 M ，则遗传操作可从这 M 个个体中生成和检测 M^3 个模式，并在此基础上能够不断形成和优化积木块，直到找到最优解。

遗传算法

1. 将目标函数映射成适应度函数的方法

● 若目标函数为最大化问题, 则 $Fit(f(x)) = f(x)$

● 若目标函数为最小化问题, 则 $Fit(f(x)) = \frac{1}{f(x)}$

↓ 将目标函数转换为求最大值的形式, 且保证函数值非负!

● 若目标函数为最大化问题, 则

$$Fit(f(x)) = \begin{cases} f(x) - C_{\min} & f(x) > C_{\min} \\ 0 & \text{其他情况} \end{cases}$$

● 若目标函数为最小化问题, 则

$$Fit(f(x)) = \begin{cases} C_{\max} - f(x) & f(x) < C_{\max} \\ 0 & \text{其他情况} \end{cases}$$

遗传算法

- 在遗传算法中，将所有妨碍适应度值高的个体产生，从而影响遗传算法正常工作的问题统称为**欺骗问题** (deceptive problem) 。
- **过早收敛**：缩小这些个体的适应度，以降低这些超级个体的竞争力。
- **停滞现象**：改变原始适应值对应的比例关系，以提高个体之间的竞争力。
- **尺度变换** (fitness scaling) 或**定标**：对适应度函数值域的某种映射变换。

遗传算法

2. 适应度函数的尺度变换

(1) 线性变换:

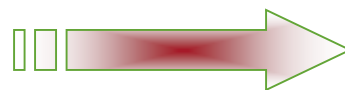
$$f' = af + b$$

满足 $f'_{avg} = f_{avg}$, $f'_{max} = C_{mult} \cdot f_{avg}$

$$a = \frac{(C_{mult} - 1)f_{avg}}{f_{max} - f_{avg}}$$

$$b = \frac{(f_{max} - C_{mult}f_{avg})f_{avg}}{f_{max} - f_{avg}}$$

满足最小适应度值非负



$$a = \frac{f_{avg}}{f_{avg} - f_{min}}$$

$$b = \frac{-f_{min}f_{avg}}{f_{avg} - f_{min}}$$

遗传算法

2. 适应度函数的尺度变换

(2) 幂函数变换法: $f' = f^K$

(3) 指数变换法: $f' = e^{-af}$

遗传算法

1. 个体选择概率分配方法

- 选择操作也称为复制 (reproduction) 操作：从当前群体中按照一定概率选出优良的个体，使它们有机会作为父代繁殖下一代子孙。
- 判断个体优良与否的准则是各个个体的适应度值：个体适应度越高，其被选择的机会就越多。

遗传算法

1. 个体选择概率分配方法

(1) 适应度比例方法 (fitness proportional model) 或蒙特卡罗法 (Monte Carlo)

● 各个个体被选择的概率和其适应度值成比例。

● 个体 i 被选择的概率为：
$$p_{si} = \frac{f_i}{\sum_{i=1}^M f_i}$$

遗传算法

1. 个体选择概率分配方法

(2) 排序方法 (rank-based model)

① 线性排序: J. E. Baker

➤ 群体成员按适应值大小从好到坏依次排列: x_1, x_2, \dots, x_N

➤ 个体 x_i 分配选择概率 p_i

$$p_i = \frac{a - bi}{M(M + 1)}$$

➤ 按转盘式选择的方式选择父体

遗传算法

1. 个体选择概率分配方法

(2) 排序方法 (rank-based model)

② 非线性排序: Z. Michalewicz

● 将群体成员按适应值从好到坏依次排列，并按下式分配选择概率：

$$p_i = \begin{cases} q(1-q)^{i-1} & i = 1, 2, \dots, M-1 \\ (1-q)^{M-1} & i = M \end{cases}$$

遗传算法

1. 个体选择概率分配方法

(2) 排序方法 (rank-based model)

● 可用其他非线性函数来分配选择概率，只要满足以下条件：

(1) 若 $P = \{x_1, x_2, \dots, x_M\}$ 且 $f(x_1) \geq f(x_2) \geq \dots \geq f(x_M)$, 则 p_i 满足

$$p_1 \geq p_2 \geq \dots \geq p_M$$

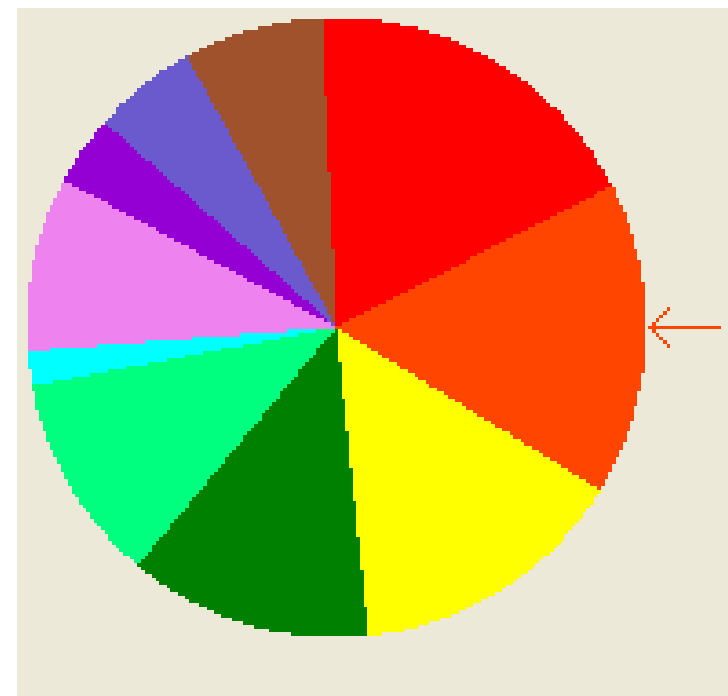
$$(2) \sum_{i=1}^M p_i = 1$$

遗传算法

2. 选择个体方法

(1) 转盘赌选择

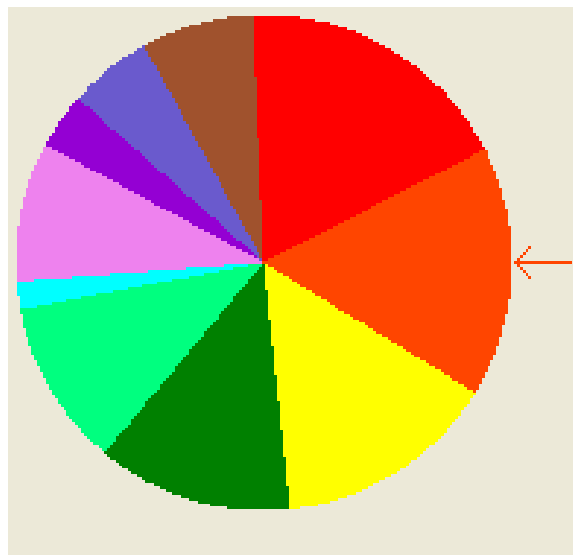
- 按个体的选择概率产生一个转盘，转盘每个区的角度与个体的选择概率成比例。
- 产生一个随机数，它落入转盘的哪个区域就选择相应的个体交叉。



遗传算法

2. 选择个体方法

(1) 转盘赌选择



个体	1	2	3	4	5	6	7	8	9	10	11
适应度	2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2	0.1
选择概率	0.18	0.16	0.15	0.13	0.11	0.09	0.07	0.06	0.03	0.02	0.0
累积概率	0.18	0.34	0.49	0.62	0.73	0.82	0.89	0.95	0.98	1.00	1.00

第1轮产生一个随机数: 0.81

第2轮产生一个随机数: 0.32

遗传算法

2. 选择个体方法

(2) 锦标赛选择方法 (tournament selection model)

● 锦标赛选择方法：从群体中随机选择个个体，将其中适应度最高的个体保存到下一代。这一过程反复执行，直到保存到下一代的个体数达到预先设定的数量为止。

● 随机竞争方法 (stochastic tournament)：每次按赌轮选择方法选取一对个体，然后让这两个个体进行竞争，适应度高者获胜。如此反复，直到选满为止。

遗传算法

1. 基本的交叉算子

(1) 一点交叉 (single-point crossover)

● 一点交叉：在个体串中随机设定一个交叉点，实行交叉时，该点前或后的两个个体的部分结构进行互换，并生成两个新的个体。

(2) 二点交叉 (two-point crossover)

● 二点交叉：随机设置两个交叉点，将两个交叉点之间的码串相互交换。

遗传算法

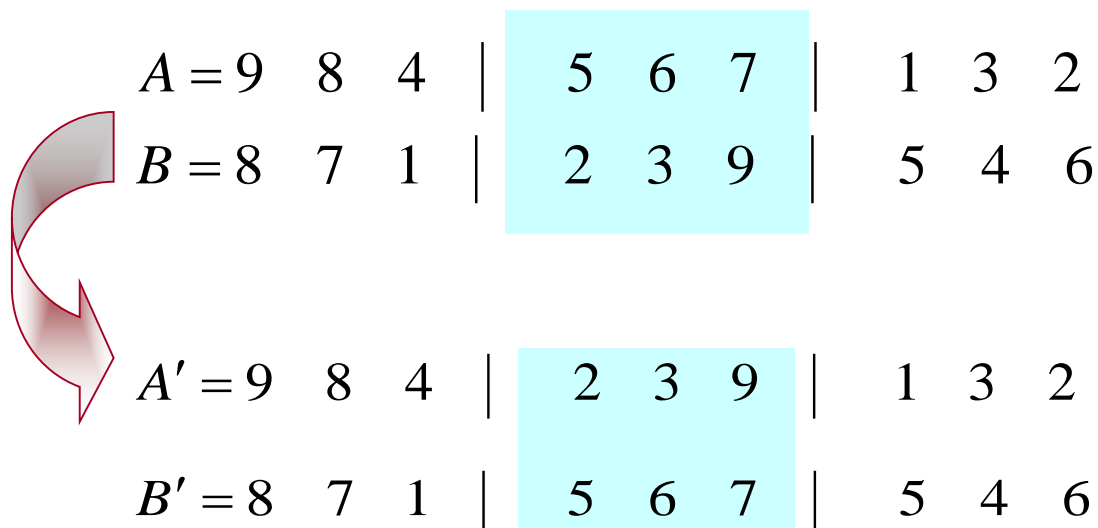
1. 基本的交叉算子

(1) 一点交叉 (single-point crossover)

● 一点交叉：在个体串中随机设定一个交叉点，实行交叉时，该点前或后的两个个体的部分结构进行互换，并生成两个新的个体。

(2) 二点交叉 (two-point crossover)

● 二点交叉：随机设置两个交叉点，将两个交叉点之间的码串相互交换。



遗传算法

- **位点变异**：群体中的个体码串，随机挑选一个或多个基因座，并对这些基因座的基因值以变异概率作变动。
- **逆转变异**：在个体码串中随机选择两点（逆转点），然后将两点之间的基因值以逆向排序插入到原位置中。
- **插入变异**：在个体码串中随机选择一个码，然后将此码插入随机选择的插入点中间。
- **互换变异**：随机选取染色体的两个基因进行简单互换。
- **移动变异**：随机选取一个基因，向左或者向右移动一个随机位数。

遗传算法

procedure: Simple GA

input: GA parameters

output: best solution

begin

$t \leftarrow 0$; // t : generation number

initialize $P(t)$ by encoding routine 编码; // $P(t)$: population of chromosomes

fitness $eval(P)$ by decoding routine 解码; 初代种群产生

while (not termination condition) do

crossover 交叉 $P(t)$ to yield $C(t)$; // $C(t)$: offspring

mutation 变异 $P(t)$ to yield $C(t)$;

fitness 适应度计算 $eval(C)$ by decoding routine 解码;

select 选择子代 $P(t+1)$ from $P(t)$ and $C(t)$;

$t \leftarrow t+1$;

end

output best solution;

end

遗传算法一般步骤

(1) 使用随机方法或者其它方法，产生一个有 N 个染色体的初始群体 $P(1)$ ， $t := 1$;

(2) 对群体中的每一个染色体 $P_i(t)$ ，计算其适应值

$$f_i = \text{fitness}(P_i(t))$$

(3) 若满足停止条件，则算法停止；否则，以概率 $p_i = f_i / \sum_{j=1}^N f_j$

从 $P(t)$ 中随机选择一些染色体构成一个新种群

$$\text{new}P(t+1) = \{P_j(t) | j = 1, 2, \dots, N\}$$

(4) 以概率 p_c 进行交叉产生一些新的染色体，得到一个新的群体

$$\text{cross}P(t+1)$$

(5) 以一个较小的概率 p_m 使染色体的一个基因发生变异，形成 $\text{mut}P(t+1)$ ； $t := t + 1$ ，成为一个新的群体 $P(t) = \text{mut}P(t+1)$

返回 (2)。

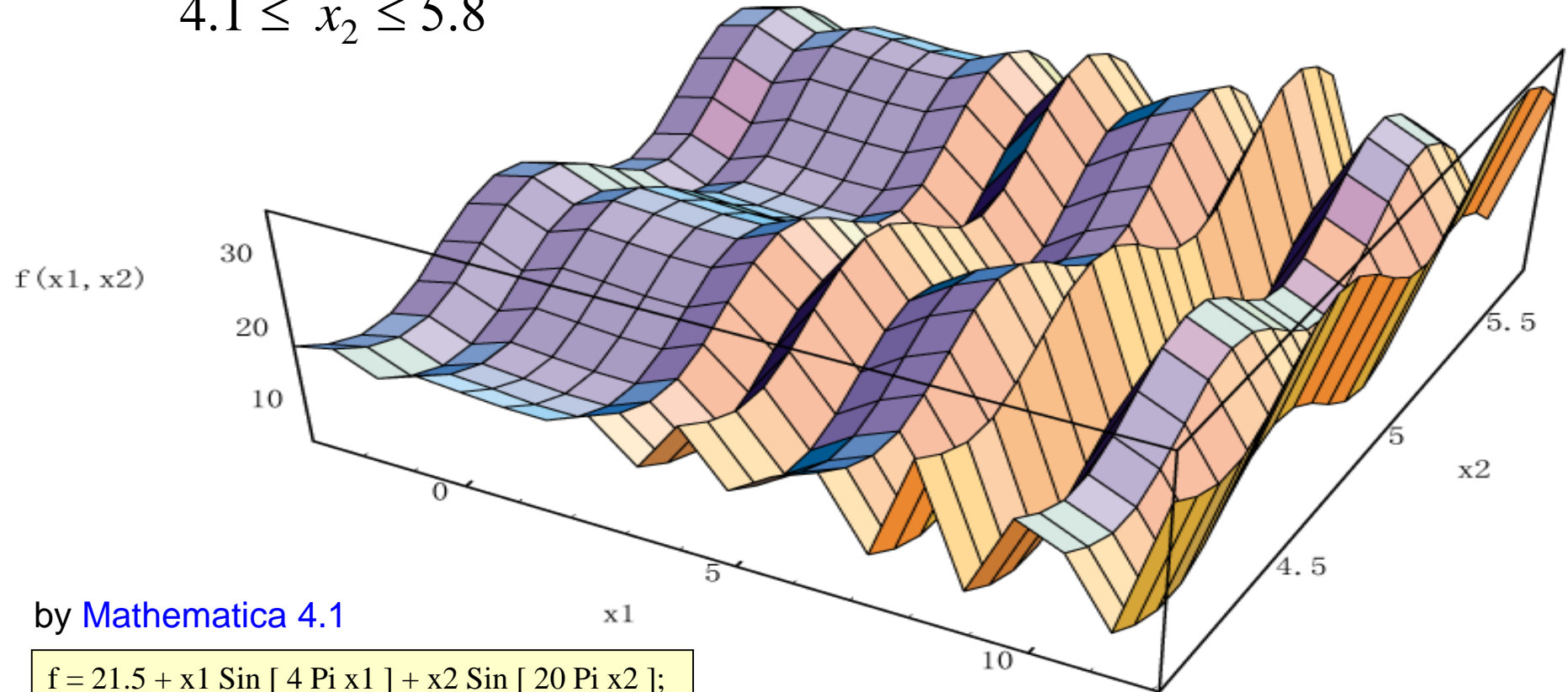
遗传算法案例分析

- We explain in detail about how a genetic algorithm actually works with a simple examples.
- We follow the approach of implementation of genetic algorithms given by Michalewicz.
 - Michalewicz, Z.: *Genetic Algorithm + Data Structures = Evolution Programs*. 3rd ed., Springer-Verlag: New York, 1996.
- The numerical example of unconstrained optimization problem is given as follows:

$$\begin{array}{ll} \max & f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2) \\ \text{s. t.} & -3.0 \leq x_1 \leq 12.1 \\ & 4.1 \leq x_2 \leq 5.8 \end{array}$$

2 Example with Simple Genetic Algorithms

$$\begin{aligned} \max \quad & f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2) \\ \text{s. t.} \quad & -3.0 \leq x_1 \leq 12.1 \\ & 4.1 \leq x_2 \leq 5.8 \end{aligned}$$



by [Mathematica 4.1](#)

```
f = 21.5 + x1 Sin [ 4 Pi x1 ] + x2 Sin [ 20 Pi x2 ];  
Plot3D[f, {x1, -3, 12.1}, {x2, 4.1, 5.8},  
  PlotPoints -> 19,  
  AxesLabel -> {x1, x2, "f(x1, x2)"}];
```

2.1 Representation

□ Binary String Representation

- The domain of x_j is $[a_j, b_j]$ and the required precision is five places after the decimal point.
- The precision requirement implies that the range of domain of each variable should be divided into at least $(b_j - a_j) \times 10^5$ size ranges.
- The required bits (denoted with m_j) for a variable is calculated as follows:

$$2^{m_j-1} < (b_j - a_j) \times 10^5 \leq 2^{m_j} - 1$$

- The mapping from a binary string to a real number for variable x_j is completed as follows:

$$x_j = a_j + \text{decimal}(\text{substring}_j) \times \frac{b_j - a_j}{2^{m_j} - 1}$$

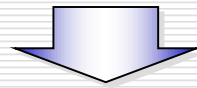
2.1 Representation

□ Binary String Encoding

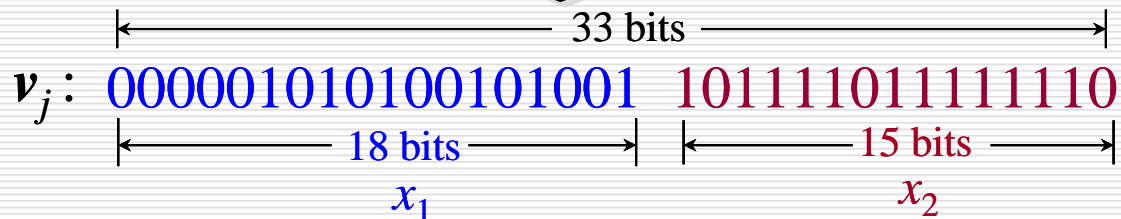
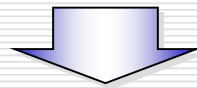
- The **precision requirement** implies that the range of domain of each variable should be divided into at least $(b_j - a_j) \times 10^5$ size ranges.
- The **required bits** (denoted with m_j) for a variable is calculated as follows:

$$x_1 : (12.1 - (-3.0)) \times 10,000 = 151,000 \\ 2^{17} < 151,000 \leq 2^{18}, \quad m_1 = 18 \text{ bits}$$

$$x_2 : (5.8 - 4.1) \times 10,000 = 17,000 \\ 2^{14} < 17,000 \leq 2^{15}, \quad m_2 = 15 \text{ bits}$$



precision requirement: $m = m_1 + m_2 = 18 + 15 = 33$ bits



2.1 Representation

□ Procedure of Binary String Encoding

input: domain of $x_j \in [a_j, b_j]$, ($j=1,2$)

output: chromosome v

step 1: The domain of x_j is $[a_j, b_j]$ and the required precision is five places after the decimal point.

step 2: The precision requirement implies that the range of domain of each variable should be divided into at least $(b_j - a_j) \times 10^5$ size ranges.

step 3: The required bits (denoted with m_j) for a variable is calculated as follows:

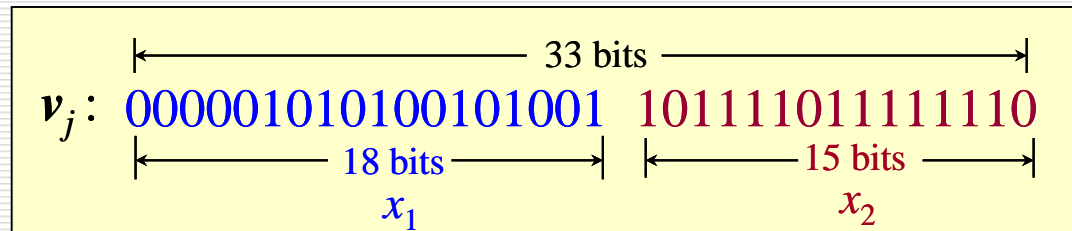
$$2^{m_j-1} < (b_j - a_j) \times 10^5 \leq 2^{m_j} - 1$$

step 4: A chromosome v is randomly generated, which has the number of genes m , where m is sum of m_j ($j=1,2$).

2.1 Representation

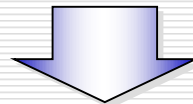
□ Binary String Decoding

- The mapping from a binary string to a real number for variable x_j is completed as follows:



	Binary Number	Decimal Number
x_1	000001010100101001	5417
x_2	1011110111111110	24318

$$x_j = a_j + \text{decimal}(\text{substring}_j) \times \frac{b_j - a_j}{2^{m_j} - 1}$$



$$x_1 = -3.0 + 5417 \times \frac{12.1 - (-3.0)}{2^{18} - 1}$$

$$= -2.687069$$

$$x_2 = 4.1 + 24318 \times \frac{5.8 - 4.1}{2^{15} - 1}$$

$$= 5.361653$$

2.1 Representation

□ Procedure of Binary String Decoding

input: substring_{*j*}

output: a real number x_j

step 1: Convert a substring (a binary string) to a decimal number.

step 2: The [mapping](#) for variable x_j is completed as follows:

$$x_j = a_j + \text{decimal}(\text{substring}_j) \times \frac{b_j - a_j}{2^{m_j} - 1}$$

2.2 Initial Population

□ Initial population is randomly generated as follows:

$$\mathbf{v}_1 = [00000101010010100110111101111110] = [x_1 \ x_2] = [-2.687969 \ 5.361653]$$

$$\mathbf{v}_2 = [001110101110011000000010101001000] = [x_1 \ x_2] = [0.474101 \ 4.170144]$$

$$\mathbf{v}_3 = [111000111000001000010101001000110] = [x_1 \ x_2] = [10.419457 \ 4.661461]$$

$$\mathbf{v}_4 = [100110110100101101000000010111001] = [x_1 \ x_2] = [6.159951 \ 4.109598]$$

$$\mathbf{v}_5 = [000010111101100010001110001101000] = [x_1 \ x_2] = [-2.301286 \ 4.477282]$$

$$\mathbf{v}_6 = [1111101010110110000000010110011001] = [x_1 \ x_2] = [11.788084 \ 4.174346]$$

$$\mathbf{v}_7 = [110100010011111000100110011101101] = [x_1 \ x_2] = [9.342067 \ 5.121702]$$

$$\mathbf{v}_8 = [001011010100001100010110011001100] = [x_1 \ x_2] = [-0.330256 \ 4.694977]$$

$$\mathbf{v}_9 = [111110001011101100011101000111101] = [x_1 \ x_2] = [11.671267 \ 4.873501]$$

$$\mathbf{v}_{10} = [111101001110101010000010101101010] = [x_1 \ x_2] = [11.446273 \ 4.171908]$$

2.3 Evaluation

- The process of **evaluating the fitness** of a chromosome consists of the following three steps:

input: chromosome v_k , $k=1, 2, \dots, popSize$

output: the fitness $eval(v_k)$

step 1: Convert the chromosome's genotype to its phenotype, *i.e.*, convert binary string into relative real values $x_k=(x_{k1}, x_{k2})$, $k = 1, 2, \dots, popSize$.

step 2: Evaluate the objective function $f(\mathbf{x}_k)$, $k = 1, 2, \dots, popSize$.

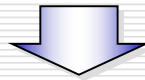
step 3: Convert the value of objective function into fitness. For the maximization problem, the fitness is simply equal to the value of objective function:

$$eval(\mathbf{v}_k) = f(\mathbf{x}_k), \quad k = 1, 2, \dots, popSize.$$

$$eval(v_k) = f(x_i) \quad (k = 1, 2, \dots, popSize)$$

$$(i = 1, 2, \dots, n)$$

$$f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2)$$



Example: $(x_1=-2.687969, x_2= 5.361653)$

$$eval(\mathbf{v}_1) = f(-2.687969, 5.361653) = 19.805119$$

2.3 Evaluation

- An evaluation function plays the role of the environment, and it rates chromosomes in terms of **their fitness**.
- The fitness function values of above chromosomes are as follows:
 $eval(\mathbf{v}_1) = f(-2.687969, 5.361653) = 19.805119$
 $eval(\mathbf{v}_2) = f(0.474101, 4.170144) = 17.370896$
 $eval(\mathbf{v}_3) = f(10.419457, 4.661461) = 9.590546$
 $eval(\mathbf{v}_4) = f(6.159951, 4.109598) = 29.406122$
 $eval(\mathbf{v}_5) = f(-2.301286, 4.477282) = 15.686091$
 $eval(\mathbf{v}_6) = f(11.788084, 4.174346) = 11.900541$
 $eval(\mathbf{v}_7) = f(9.342067, 5.121702) = 17.958717$
 $eval(\mathbf{v}_8) = f(-0.330256, 4.694977) = 19.763190$
 $eval(\mathbf{v}_9) = f(11.671267, 4.873501) = 26.401669$
 $eval(\mathbf{v}_{10}) = f(11.446273, 4.171908) = 10.252480$
- It is clear that chromosome \mathbf{v}_4 is the strongest one and that chromosome \mathbf{v}_3 is the weakest one.

2.4 Genetic Operators

□ Selection:

- In most practices, a roulette wheel approach is adopted as the selection procedure, which is one of the fitness-proportional selection and can select a new population with respect to the probability distribution based on fitness values.
- The roulette wheel can be constructed with the following steps:

input: population $P(t-1)$, $C(t-1)$

output: population $P(t)$, $C(t)$

step 1: Calculate the total fitness for the population $F = \sum_{k=1}^{popSize} eval(\mathbf{v}_k)$

step 2: Calculate selection probability p_k for each chromosome \mathbf{v}_k

$$p_k = \frac{eval(\mathbf{v}_k)}{F}, \quad k = 1, 2, \dots, popSize$$

step 3: Calculate cumulative probability q_k for each chromosome \mathbf{v}_k

$$q_k = \sum_{j=1}^k p_j, \quad k = 1, 2, \dots, popSize$$

step 4: Generate a random number r from the range $[0, 1]$.

step 5: If $r \leq q_1$, then select the first chromosome \mathbf{v}_1 ; otherwise, select the k th chromosome \mathbf{v}_k ($2 \leq k \leq popSize$) such that $q_{k-1} < r \leq q_k$.

2.4 Genetic Operators

□ Illustration of Selection:

input: population $P(t-1)$, $C(t-1)$

output: population $P(t)$, $C(t)$

step 1: Calculate the total fitness F for the population.

$$F = \sum_{k=1}^{10} eval(\mathbf{v}_k) = 178.135372$$

step 2: Calculate selection probability p_k for each chromosome \mathbf{v}_k .

$$\begin{aligned} p_1 &= 0.111180, & p_2 &= 0.097515, & p_3 &= 0.053839, & p_4 &= 0.165077, \\ p_5 &= 0.088057, & p_6 &= 0.066806, & p_7 &= 0.100815, & p_8 &= 0.110945, \\ p_9 &= 0.148211, & p_{10} &= 0.057554 \end{aligned}$$

step 3: Calculate cumulative probability q_k for each chromosome \mathbf{v}_k .

$$\begin{aligned} q_1 &= 0.111180, & q_2 &= 0.208695, & q_3 &= 0.262534, & q_4 &= 0.427611, \\ q_5 &= 0.515668, & q_6 &= 0.582475, & q_7 &= 0.683290, & q_8 &= 0.794234, \\ q_9 &= 0.942446, & q_{10} &= 1.000000 \end{aligned}$$

step 4: Generate a random number r from the range $[0,1]$.

$$\begin{array}{ccccc} 0.301431, & 0.322062, & 0.766503, & 0.881893, & 0.350871, \\ 0.583392, & 0.177618, & 0.343242, & 0.032685, & 0.197577 \end{array}$$

Which one would be chosen?

2.4 Genetic Operators

□ Illustration of Selection:

step 5: $q_3 < r_1 = 0.301432 \leq q_4$, it means that the chromosome \mathbf{v}_4 is selected for new population; $q_3 < r_2 = 0.322062 \leq q_4$, it means that the chromosome \mathbf{v}_4 is selected again, and so on. Finally, the new population consists of the following chromosome.

$$\mathbf{v}_1' = [100110110100101101000000010111001] \quad (\mathbf{v}_4)$$

$$\mathbf{v}_2' = [100110110100101101000000010111001] \quad (\mathbf{v}_4)$$

$$\mathbf{v}_3' = [001011010100001100010110011001100] \quad (\mathbf{v}_8)$$

$$\mathbf{v}_4' = [111110001011101100011101000111101] \quad (\mathbf{v}_9)$$

$$\mathbf{v}_5' = [100110110100101101000000010111001] \quad (\mathbf{v}_4)$$

$$\mathbf{v}_6' = [110100010011111000100110011101101] \quad (\mathbf{v}_7)$$

$$\mathbf{v}_7' = [0011101011100110000000010101001000] \quad (\mathbf{v}_2)$$

$$\mathbf{v}_8' = [100110110100101101000000010111001] \quad (\mathbf{v}_4)$$

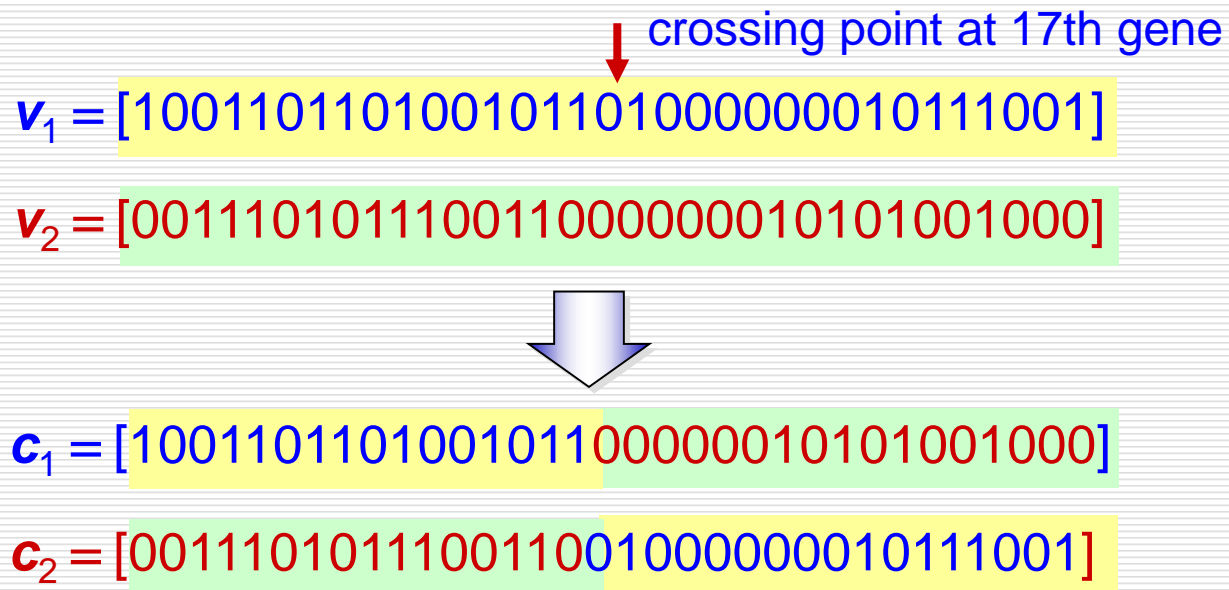
$$\mathbf{v}_9' = [00000101010010100110111101111110] \quad (\mathbf{v}_1)$$

$$\mathbf{v}_{10}' = [0011101011100110000000010101001000] \quad (\mathbf{v}_2)$$

2.4 Genetic Operators

□ Crossover (One-cut point Crossover)

- Crossover used here is one-cut point method, which random selects one cut point.
- **Exchanges** the right parts of two parents to generate offspring.
- Consider two chromosomes as follow and the cut point is randomly selected after the 17th gene:



2.4 Genetic Operators

□ Procedure of One-cut Point Crossover:

procedure: One-cut Point Crossover

input: p_c , parent P_k , $k=1, 2, \dots, popSize$

output: offspring C_k

begin

for $k \leftarrow 1$ **to** $\lceil popSize/2 \rceil$ **do** // $popSize$: population size

if $p_c \geq \text{random}[0, 1]$ **then** // p_c : the probability of crossover

$i \leftarrow 0$;

$j \leftarrow 0$;

repeat

$i \leftarrow \text{random}[1, popSize]$;

$j \leftarrow \text{random}[1, popSize]$;

until ($i \neq j$)

$p \leftarrow \text{random}[1, l-1]$; // p : the cut position, l : the length of chromosome

$C_i \leftarrow P_i[1: p-1] \parallel P_j[p: l]$;

$C_j \leftarrow P_j[1: p-1] \parallel P_i[p: l]$;

end

end

output offspring C_k ;

end

2.4 Genetic Operators

□ Mutation

- Alters one or more genes with a probability equal to the mutation rate.
- Assume that the 16th gene of the chromosome v_1 is selected for a mutation.
- Since the gene is 1, it would be flipped into 0. So the chromosome after mutation would be:

mutating point at 16th gene
↓
 $v_1 = [100110110100101101000000010111001]$
↓
 $c_1 = [100110110100101000000010101001000]$

2. Example with Simple Genetic Algorithms

□ Procedure of Mutation:

```
procedure: Mutation
input:  $p_M$ , parent  $P_k$ ,  $k=1, 2, \dots, popSize$ 
output: offspring  $C_k$ 
begin
  for  $k \leftarrow 1$  to  $popSize$  do           //  $popSize$ : population size
    for  $j \leftarrow 1$  to  $l$  do             //  $l$ : the length of chromosome
      if  $p_M \geq \text{random}[0, 1]$  then      //  $p_M$ : the probability of mutation
         $C_k \leftarrow P_k[1:j-1] // P_k[j]^* p // P_k[j+1:l]$ ;
      end
    end
  end
  output offspring  $C_k$ ;
end
```

□ Illustration of Mutation:

Assume that $p_M = 0.01$

<i>chromNum</i>	<i>bitNo</i>	<i>randomNum</i>
4	6	0.009857
5	32	0.003113
7	1	0.000946
8	3	0.011282
10	32	0.001282

$> p_M = 0.01$

2. Example with Simple Genetic Algorithms

□ Next Generation

$$\begin{aligned} \mathbf{v}_1' &= [100110110100101101000000010111001], & f(6.159951, 4.109598) &= 29.406122 \\ \mathbf{v}_2' &= [100110110100101101000000010111001], & f(6.159951, 4.109598) &= 29.406122 \\ \mathbf{v}_3' &= [001011010100001100010110011001100], & f(-0.330256, 4.694977) &= 19.763190 \\ \mathbf{v}_4' &= [111110001011101100011101000111101], & f(11.907206, 4.873501) &= 5.702781 \\ \mathbf{v}_5' &= [100110110100101101000000010111001], & f(8.024130, 4.170248) &= 19.91025 \\ \mathbf{v}_6' &= [110100010011111000100110011101101], & f(9.34067, 5.121702) &= 17.958717 \\ \mathbf{v}_7' &= [100110110100101101000000010111001], & f(6.159951, 4.109598) &= 29.406122 \\ \mathbf{v}_8' &= [100110110100101101000000010111001], & f(6.159951, 4.109598) &= 29.406122 \\ \mathbf{v}_9' &= [000001010100101001101111011111110], & f(-2.687969, 5.361653) &= 19.805199 \\ \mathbf{v}_{10}' &= [0011101011100110000000010101001000], & f(0.474101, 4.170248) &= 17.370896 \end{aligned}$$

2. Example with Simple Genetic Algorithms

□ Procedure of GA for Unconstrained Optimization

```
procedure: GA for Unconstrained Optimization (uO)
input: uO data set, GA parameters
output: best solution
begin
     $t \leftarrow 0$ ;
    initialize  $P(t)$  by binary string encoding;
    fitness  $eval(P)$  by binary string decoding;
    while (not termination condition) do
        crossover  $P(t)$  to yield  $C(t)$  by one-cut point crossover;
        mutation  $P(t)$  to yield  $C(t)$ ;
        fitness  $eval(C)$  by binary string decoding;
        select  $P(t+1)$  from  $P(t)$  and  $C(t)$  by roulette wheel selection;
         $t \leftarrow t+1$ ;
    end
    output best solution;
end
```

2. Example with Simple Genetic Algorithms

□ Final Result

- The test run is terminated after 1000 generations.
- We obtained the best chromosome in the 884th generation as follows:

$$\max \quad f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2)$$

$$\text{s. t.} \quad -3.0 \leq x_1 \leq 12.1$$

$$4.1 \leq x_2 \leq 5.8$$

$$\begin{aligned} eval(\mathbf{v}^*) &= f(11.622766, 5.624329) \\ &= 38.737524 \end{aligned}$$

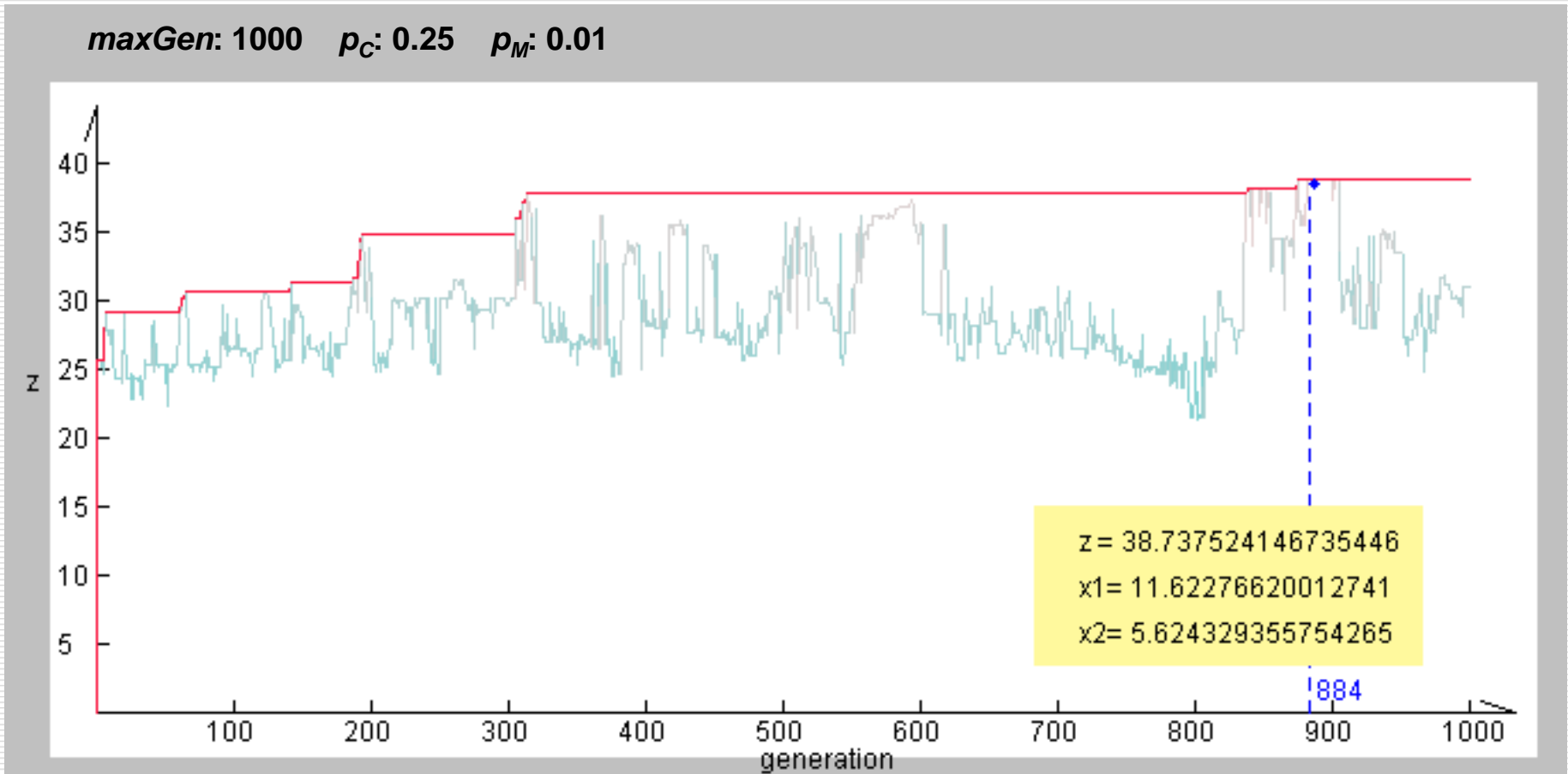
$$x_1^* = 11.622766$$

$$x_2^* = 5.624329$$

$$f(x_1^*, x_2^*) = 38.737524$$

2. Example with Simple Genetic Algorithms

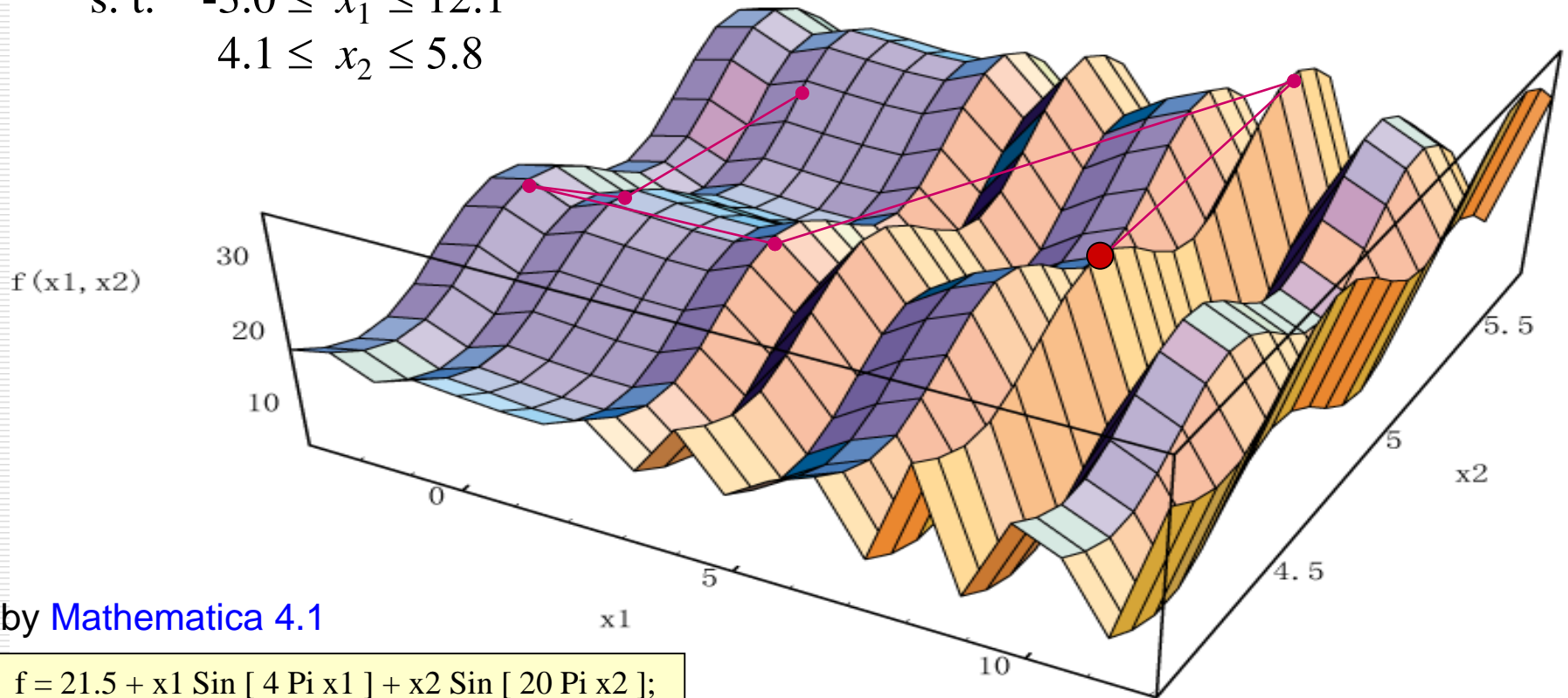
□ Evolutional Process



2. Example with Simple Genetic Algorithms

□ Evolutional Process

$$\begin{aligned} \max \quad & f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2) \\ \text{s. t.} \quad & -3.0 \leq x_1 \leq 12.1 \\ & 4.1 \leq x_2 \leq 5.8 \end{aligned}$$



by [Mathematica 4.1](#)

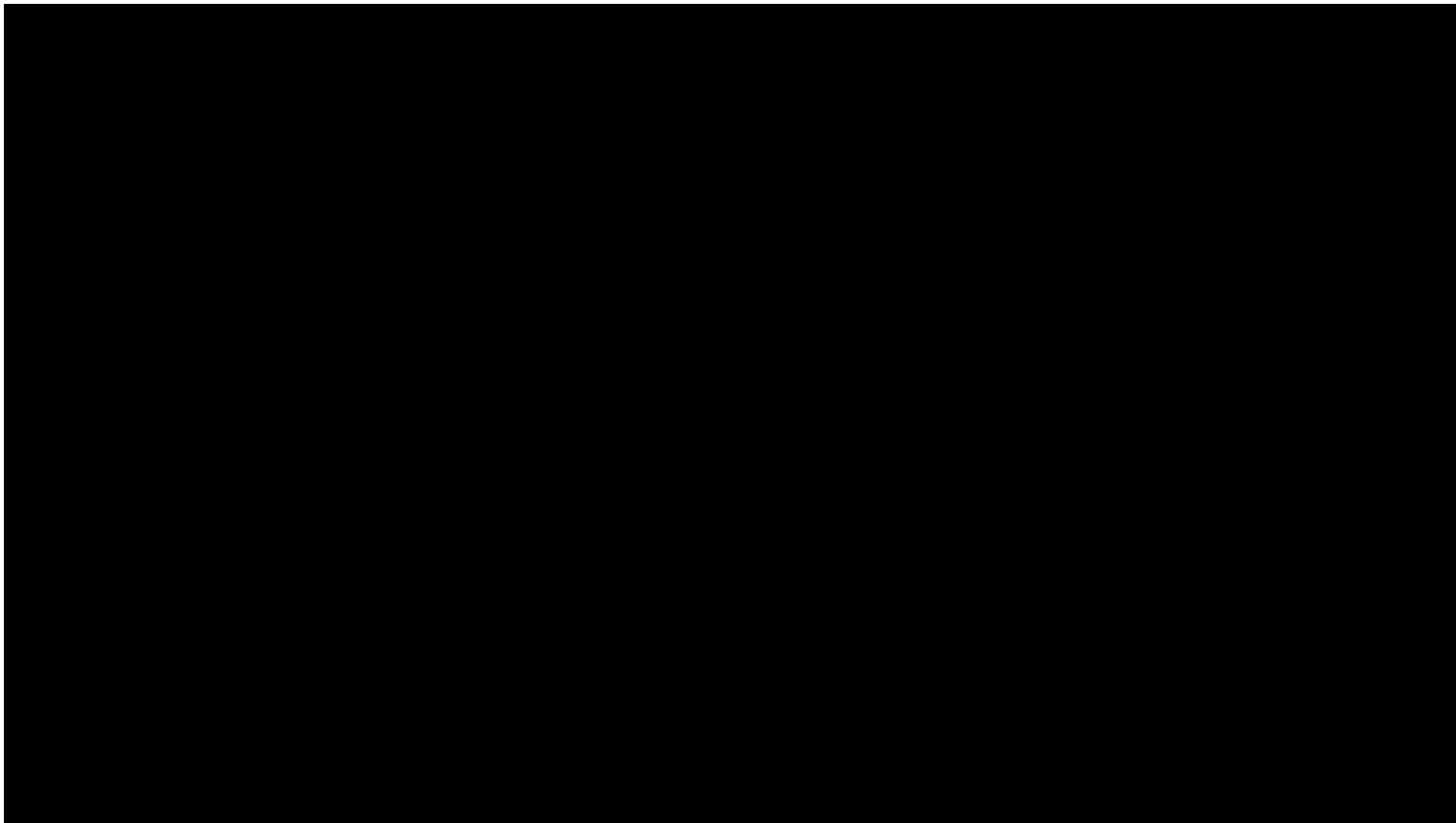
```
f = 21.5 + x1 Sin [ 4 Pi x1 ] + x2 Sin [ 20 Pi x2 ];  
Plot3D[f, {x1, -3.0, 12.1}, {x2, 4.1, 5.8},  
  PlotPoints -> 19,  
  AxesLabel -> {x1, x2, "f(x1, x2)"}];  
ContourPlot[ f, {x, -3.0, 12.1}, {y, 4.1, 5.8}];
```

遗传算法的特点

遗传算法是一种全局优化概率算法，**主要特点**有：

- ◆ 遗传算法对所求解的优化问题没有太多的数学要求，由于进化特性，搜索过程中不需要问题的内在性质，可直接对结构对象进行操作。
- ◆ 利用随机技术指导对一个被编码的参数空间进行高效率搜索
- ◆ 采用群体搜索策略，易于并行化。
- ◆ 仅用适应度函数值来评估个体，并在此基础上进行遗传操作，使种群中个体之间进行信息交换。
- ◆ 遗传算法能够非常有效地进行概率意义的全局搜索。

遗传算法应用



高级搜索

- 爬山算法
- 模拟退火算法
- 遗传算法
- 应用案例

3. Traveling Salesman Problem

- The **Traveling Salesman Problem (TSP)** is one of the most widely studied combinatorial optimization problems.
- Its statement is deceptively simple: A salesperson **seeks the shortest tour through n cities**.

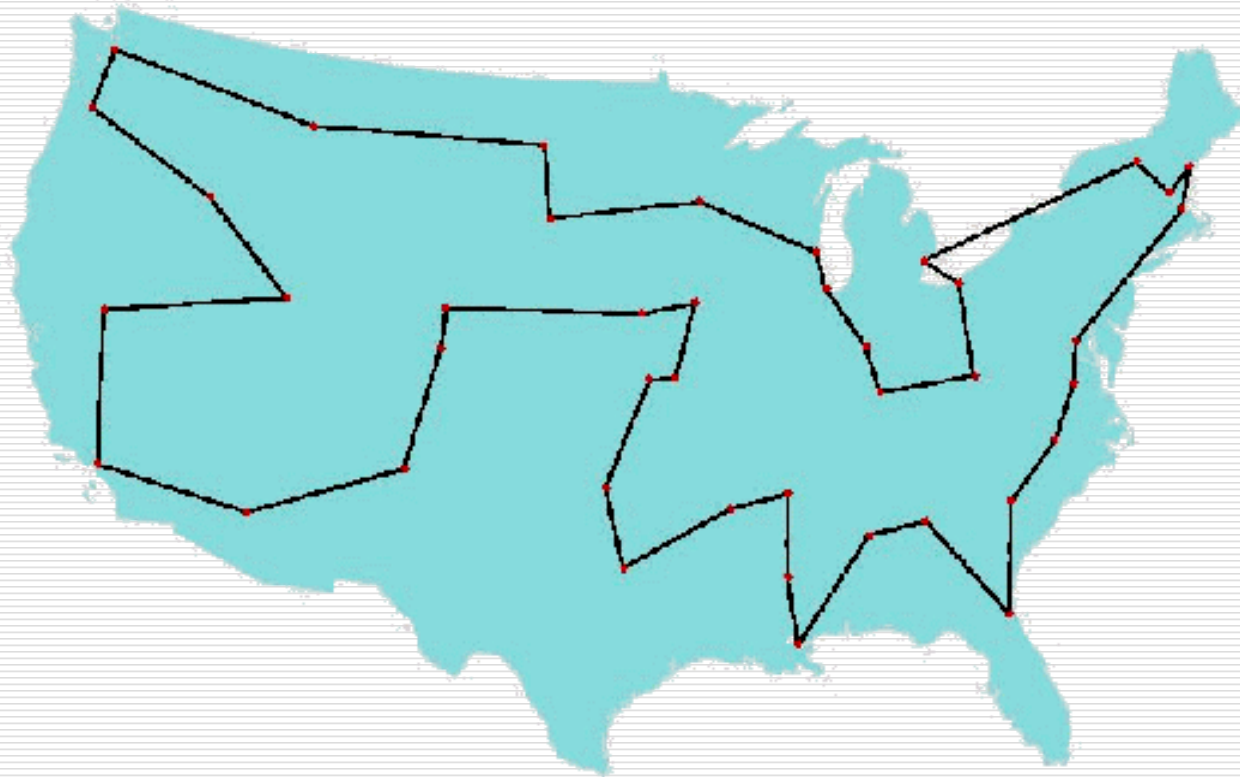


Fig. 3.4 **George Dantzig, Ray Fulkerson, and Selmer Johnson (1954)**
a description of a method for solving the TSP :49 cities

3. Traveling Salesman Problem

❑ Existing Instances

■ 49 city problem

- ❑ Dagtzig, G., D. Fulkerson and S. Johnson: “Solution of a large scale traveling salesman problems”, *Operations Research*, vol. 2, pp. 393-410, 1954.

■ 120 city problem

- ❑ Grötschel, M.: “On the symmetric traveling salesman problem: solution of a 120 city problem”, *Mathematical Programming Studies*, vol. 12, pp. 61-77, 1980.

■ 318 city problem

- ❑ Crowder, H. and M. Padberg: “Solving large scale symmetric traveling salesman problems to optimality”, *Management Science*, vol. 22, pp. 15-24, 1995.

■ 532 city problem

- ❑ Padberg, M. and G. Rinaldi: “Optimization of 532 city symmetric traveling salesman problem by branch and cut”, *Operations Research Letters*, vol. 6, pp. 1-7, 1987.

3. Traveling Salesman Problem

■ 666 city problem

- Grötschel, M. and O. Holland: “Solution of large scale symmetric traveling salesman problems”, *Mathematical Programming Studies*, vol. 51, pp. 141-202, 1991.

■ 2392 city problem

- Padberg, M. and G. Rinaldi: “A branch and cut algorithm for the resolution of large scale symmetric traveling salesman problem”, *SIAM Review*, vol. 33, pp. 60-100, 1991.

■ The earlier studies using the genetic algorithm to solve TSP

- Grefenstette, J.: *Proceedings of the First International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

■ TSP has become a target for the genetic algorithm community.

- Michalewicz, Z.: *Genetic Algorithm + Data structure = Evolution Programs*, 2nd ed., Springer-Verlag, New York, 1994.

6. Traveling Salesman Problem

□ Notations

■ Indices

i, j : the index of city, $i, j = 1, 2, \dots, n$

■ Parameters

n : the total number of cities

d_{ij} : the distance city i to city j , i.e., the distance of route (i, j) ; the distance matrix (d_{ij}) is symmetric.

■ Decision Variables

x_{ij} : the 0,1 decision variable; 1, if route (i, j) is selected, and 0, otherwise.

3. Traveling Salesman Problem

□ Mathematical Model of TSP

$$\min \quad z = \sum_{i=1}^n \sum_{j=1}^n (d_{ij} x_{ij} + d_{n1} x_{n1})$$

$$\text{s.t.} \quad \sum_{i=1}^n x_{ij} = 1, \quad j = 1, 2, 3, \dots, n$$

$$\sum_{j=1}^n x_{ij} = 1, \quad i = 1, 2, 3, \dots, n$$

$$x_{ij} = \begin{cases} 1, & \text{if route (i,j) is selected} \\ 0, & \text{otherwise} \end{cases}$$

data set for non-directed graph

<i>i</i>	<i>j</i>	<i>d_{ij}</i>	<i>i</i>	<i>j</i>	<i>d_{ij}</i>
1	2	8	3	7	7
	3	5		8	12
	4	9		9	12
	5	12		4	5
	6	14		6	17
	7	12		7	10
	8	16		8	7
	9	17		9	15
	2	3		5	6
2	4	15		7	10
	5	17		8	6
	6	8		9	15
	7	11	6	7	9
	8	18		8	14
	9	14		9	8
	3	4		7	8
	5	9		9	6
	6	11		8	9
	8	11		9	11
3	4	7		7	8
	5	9		9	6
	6	11		8	9
	8	11		9	11
	9	14		9	8
	2	3		5	6

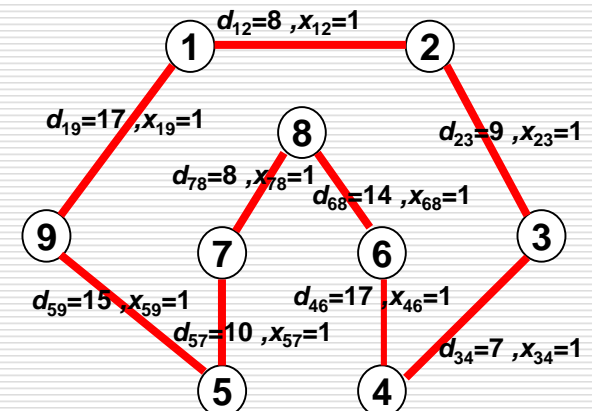


Fig. 3.5 Simple TSP Model

3.1 Representation

□ Permutation Representation

- This direct representation is perhaps the most natural representation of a TSP, where cities are listed in the order in which they are visited.

	1	2	3	4	5	6	7	8	9
chromosome 染色体	3	2	5	4	7	1	6	9	8

tour list 旅行路径 3 - 2 - 5 - 4 - 7 - 1 - 6 - 9 - 8

- This representation is also called a path representation or order representation.

procedure: Permutation Encoding

Input: city set, total number of cities N

output: chromosome v

begin

for $j=1$ to N

$v(j) \leftarrow j$;

for $i=1$ to $\lceil N/2 \rceil$

repeat

$j \leftarrow \text{random}[1, N]$;

$k \leftarrow \text{random}[1, N]$;

until $j \neq k$

$\text{swap}(v(j), v(k))$;

output chromosome v ;

end

procedure: Permutation Decoding

Input: chromosome v ,

 total number of cities N

output: tour list L

begin

$L \leftarrow \emptyset$;

for $i=1$ to N

$L \leftarrow L \cup v(i)$;

output tour list L ;

end

3.1 Representation

□ Random Keys Representation

- This indirect representation encodes a solution with random numbers from (0,1).
- These values are used as sort keys to decode the solution.

	1	2	3	4	5	6	7	8	9
chromosome	0.23	0.82	0.45	0.74	0.87	0.11	0.56	0.69	0.78

where position i in the list represents city i .

tour list 5 - 2 - 9 - 4 - 8 - 7 - 3 - 1 - 6

procedure: Random Keys Encoding

Input: city set,

total number of cities N

output: chromosome v

begin

for $i=1$ to N

$v[i] \leftarrow \text{random}[0,1];$

output chromosome v ;

end

procedure: Random Keys Decoding

Input: chromosome v ,

total number of cities N

output: tour list L

begin

$L \leftarrow \emptyset;$

for $i=1$ to N

$L \leftarrow L \cup i;$

sort L by $v[i];$

output tour list L ;

end

3.2 Crossover Operators

- During the past decade, several crossover operators have been proposed for permutation representation, such as **partial-mapped crossover (PMX)**, **order crossover (OX)**, **cycle crossover (CX)**, **position-based crossover**, **order-based crossover**, **heuristic crossover**, and so on.
- These operators can be classified into two classes:
 - **Canonical approach**
 - The canonical approach can be viewed as an extension of two-point or multipoint crossover of binary strings to permutation representation.
 - **Heuristic approach**
 - The application of heuristics in crossover intends to generate an improved offspring.

3.2 Crossover Operators

1. Partial-Mapped Crossover (PMX)

procedure : PMX crossover

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'_1, v'_2

begin

$R \leftarrow \emptyset$;

// step 1: select two positions
at random

$s \leftarrow \text{random}[1:l-1]$;

$t \leftarrow \text{random}[s+1:l]$;

// step 2: exchange two substrings

$v'_1 \leftarrow v_1[1:s-1] \parallel v_2[s:t] \parallel v_1[t+1:l]$;

$v'_2 \leftarrow v_2[1:s-1] \parallel v_1[s:t] \parallel v_2[t+1:l]$;

// step 3: determine the mapping
relationship

$R \leftarrow \text{relation}(v_1[s:t], v_2[s:t])$;

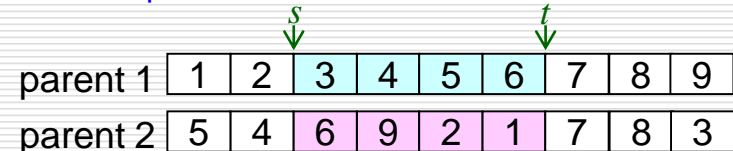
// step 4: legalize offspring

$\text{legalize}(v'_1, v'_2, R)$;

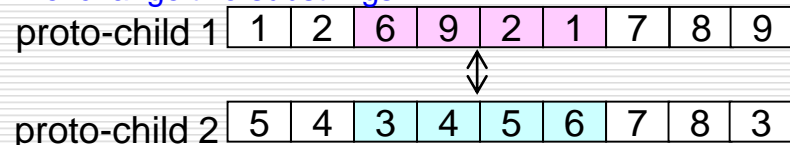
output offspring v'_1, v'_2 ;

end

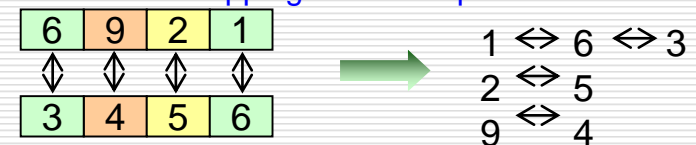
step 1 : select two positions at random



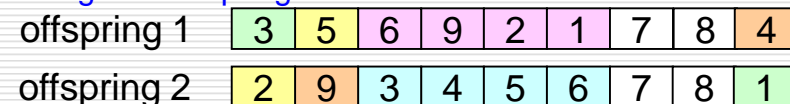
step 2: exchange two substrings



step 3 : determine the mapping relationship



step 4 : legalize offspring



v_1 : parent chromosome 1

l : length of chromosome

v'_2 : offspring chromosome 2

s : start position of substring

$\text{relation}(v_1, v_2)$: searching relationship between v_1 and v_2

$\text{legalize}(v_1, v_2, R)$ change genes value of v_1, v_2 based on relationship R

v_2 : parent chromosome 2

v'_1 : offspring chromosome 1

R : relationships

t : end position of substring

3.2 Crossover Operators

2. OX crossover

procedure : Order Crossover (OX)

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'

begin

$w \leftarrow 1$;

// step 1: select substring at random

$s \leftarrow \text{random}[1: l-1]$;

$t \leftarrow \text{random}[s+1: l]$;

// step 2: produce a proto-child by
copying the substring

$v' \leftarrow v_1[s: t]$;

// step 3: produce offspring by filling
unfixed positions of proto-
child from parent 2

for $i=1$ **to** $s-1$

for $j=w$ **to** l

$fg \leftarrow 0$;

for $k=s$ **to** t

if $v_2[j] = v_1[k]$ **then**

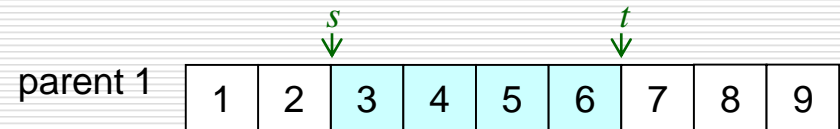
$fg \leftarrow 1$; **break**;

if $fg=0$ **then**

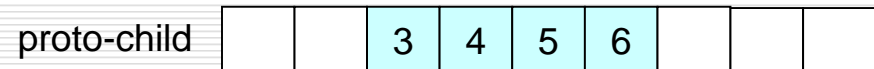
$v'[i] \leftarrow v_2[j]$;

$w \leftarrow j + 1$; **break**;

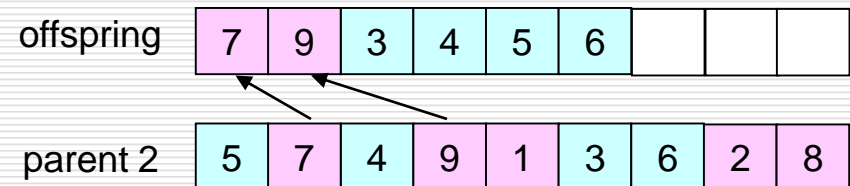
step 1 : select substring at random



step 2 : produce a proto-child by copying the substring



step 3 : produce offspring by filling unfixed positions
of proto-child from parent 2



v_1 : parent chromosome 1

l : length of chromosome

w : working data

s : start position of substring

v_2 : parent chromosome 2

v' : offspring chromosome

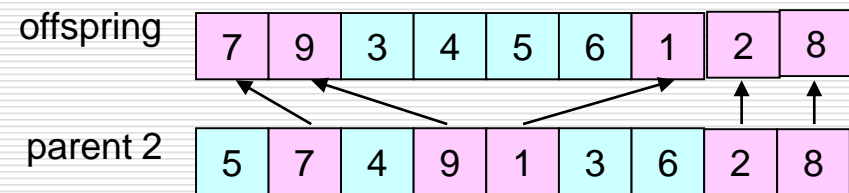
fg : flag

t : end position of substring

3.2 Crossover Operators

```
for  $i=t+1$  to  $l$ 
  for  $j=w$  to  $l$ 
     $fg \leftarrow 0$ ;
    for  $k=s$  to  $t$ 
      if  $v_2[j] = v_1[k]$  then
         $fg \leftarrow 1$ ; break;
    if  $fg=0$  then
       $v'[i] \leftarrow v_2[j]$ ;
       $w \leftarrow j+1$ ; break;
```

```
output offspring  $v'$  ;
end;
```



v_1 : parent chromosome 1	v_2 : parent chromosome 2
l : length of chromosome	v' : offspring chromosome
w : working data	fg : flag
s : start position of substring	t : end position of substring

3.2 Crossover Operators

3. Position-based Crossover (PBX)

procedure : Position-based Crossover

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'

begin

$T \leftarrow \emptyset, S \leftarrow \emptyset, w \leftarrow 1;$

// step 1: select a set of positions
from parent 1 at random

$N \leftarrow \text{random}[1:l];$

// step 2: produce proto-child by
copying values of
selected positions

for $i=1$ **to** N

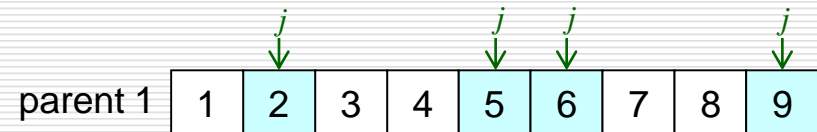
$j \leftarrow \text{random}[1:l];$

$v'[j] \leftarrow v_1[j];$

$T \leftarrow T \cup j;$

$S \leftarrow S \cup v_1[j];$

step 1 : select a set of positions from parent 1 at random



step 2 : produce proto-child by copying
values of selected positions



v_1 : parent chromosome 1 v_2 : parent chromosome 2
 l : length of chromosome v' : offspring chromosome

N : total number of selected positions

$T = \{t[j]\}, j=1, 2, \dots, N$: selected positions set

$S = \{s[m]\}, m=1, 2, \dots, N$: genes value set of selected positions

fg_1 : flag 1

fg_2 : flag 2

w : working data

3.2 Crossover Operators

// step 3: produce offspring by filing unfixed positions of proto-child from parent 2

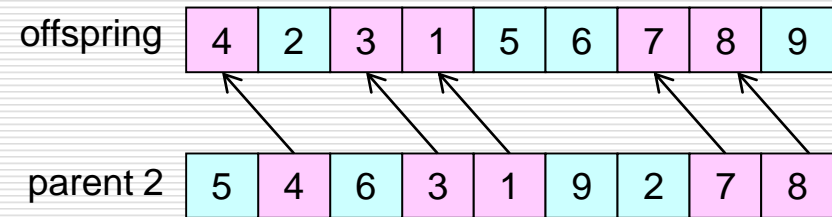
```

for i=1 to l
  fg1 ← 0;
  for j=1 to N
    if i=t[j] then fg1 ← 1;
  if fg1=1 then continue;
  for k=w to l
    fg2 ← 0;
    for m=1 to N
      if v2[k]=s[m] then
        fg2 ← 1; break;
    if fg2=0 then
      v'[i] ← v2[k];
      w ← k + 1 ; break ;

```

output offspring v' ;
end

step 3 : produce offspring by filing unfixed positions of proto-child from parent 2



v_1 : parent chromosome 1 v_2 : parent chromosome 2
 l : length of chromosome v' : offspring chromosome
 N : total number of selected positions
 $T=\{t[j]\}$, $j=1,2,\dots, N$: selected positions set
 $S=\{s[m]\}$, $m=1,2,\dots, N$: genes value set of selected positions
 fg_1 : flag 1 fg_2 : flag 2
 w : working data

3.2 Crossover Operators

4. Order-Based Crossover (OBX)

procedure : Order-Based Crossover

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'

begin

$S \leftarrow \emptyset, T \leftarrow \emptyset, w \leftarrow 1;$

// step 1: select a set of positions
from parent 1 at random

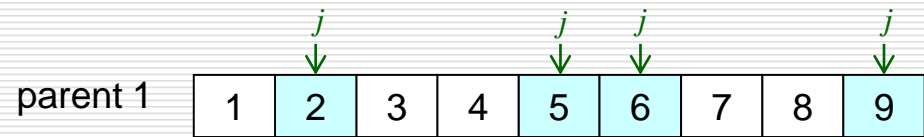
$N \leftarrow \text{random}[1:l];$

for $i=1$ **to** N

$j \leftarrow \text{random}[1:l];$

$S \leftarrow S \cup v_1[j];$

step 1 : select a set of positions
from parent 1 at random



v_1 : parent chromosome 1

v_2 : parent chromosome 2

l : length of chromosome

v' : offspring chromosome

N : total number of selected positions

$T=\{t[j]\}$, $j=1,2,\dots, N$: positions set of assigned genes from v_2 to v'

$S=\{s[i]\}$, $i=1,2,\dots, N$: genes value set of selected positions

fg_1 : flag 1

fg_2 : flag 2

w : working data

3.2 Crossover Operators

// step 2: produce proto-child by copying non-selected value from parent 2

for $i=1$ to l

$fg_1 \leftarrow 0$;

for $j=1$ to N

if $v_2[i] = s[j]$ then

$fg_1 \leftarrow 1$; break;

if $fg_1=0$ then

$v'[i] \leftarrow v_2[i]$,

$T \leftarrow T \cup i$;

// step 3: produce offspring by copying selected values from parent 1

for $i=1$ to l

$fg_2 \leftarrow 0$;

for $j=1$ to N

if $i = t[j]$ then $fg_2 \leftarrow 1$;

if $fg_2=1$ then continue;

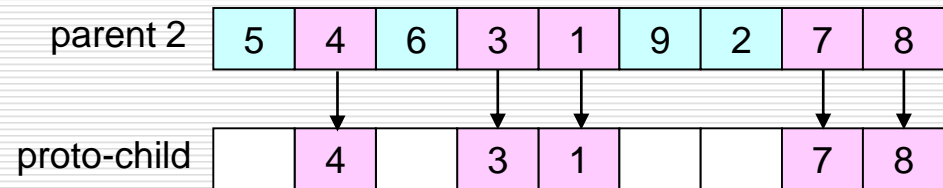
$v'[i] \leftarrow s[w]$;

$w \leftarrow w+1$;

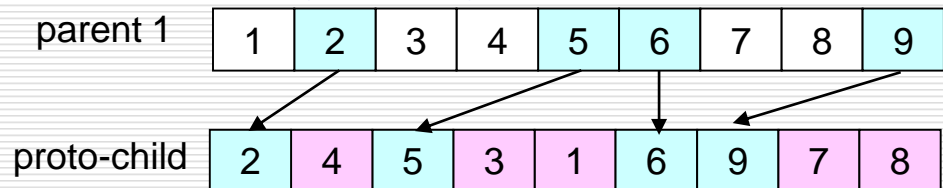
output offspring v' ;

end

step 2: produce proto-child by copying non-selected value from parent 2



step 3 : produce offspring by copying selected values from parent 1



v_1 : parent chromosome 1

v_2 : parent chromosome 2

l : length of chromosome

v' : offspring chromosome

N : total number of selected positions

$T=\{t[j]\}$, $j=1,2,\dots, N$: positions set of assigned genes from v_2 to v'

$S=\{s[i]\}$, $i=1,2,\dots, N$: genes value set of selected positions

fg_1 : flag 1

fg_2 : flag 2

w : working data

3.2 Crossover Operators

5. Cycle Crossover (CX)

procedure : CX crossover

input : chromosome v_1, v_2 ,

length of chromosome l

output : offspring v'

begin

$S \leftarrow \emptyset, T \leftarrow \emptyset, w \leftarrow 1;$

// step 1: find the cycle between parents

$C \leftarrow \text{cy}(v_1, v_2);$

// step 2: produce proto-child by copying
gene values in cycle from parent 1

for $i=1$ **to** l

for $j=1$ **to** $N-1$

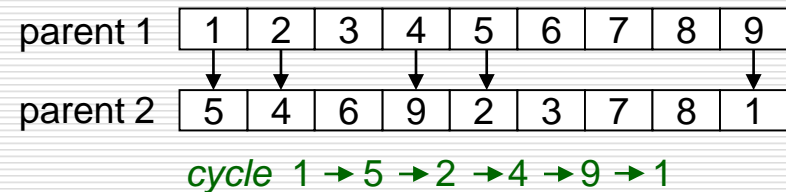
if $v_1[i] = c[j]$ **then**

$v'[i] \leftarrow v_1[i];$

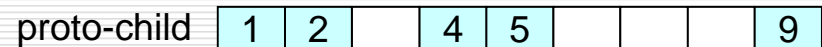
$T \leftarrow T \cup i;$

$S \leftarrow S \cup v_1[i];$

step 1 : find the cycle between parents



step 2 : produce proto-child by copying gene values
in cycle from parent 1



v_1 : parent chromosome 1 v_2 : parent chromosome 2
 l : length of chromosome v' : offspring chromosome
 N : total number of values in cycle

$T = \{t[n]\}, n=1,2,\dots, N-1$: positions set of S in proto-child

$S = \{s[k]\}, k=1,2,\dots, N-1$: proto-child genes value set in cycle

$C = \{c[j]\}, j=1,2,\dots, N-1$: value set of cycle

fg_1 : flag 1

fg_2 : flag 2

w : working data

$\text{cy}(v_1, v_2)$: finding the cycle which is defined

by the corresponding positions between parents

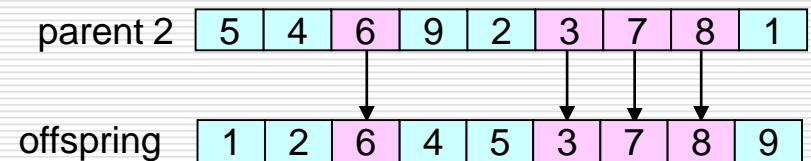
3.2 Crossover Operators

// step 3: produce offspring by filling
unfixed position from parent 2

```

for  $i=1$  to  $l$ 
     $fg_1 \leftarrow 0$ ;
    for  $n=1$  to  $|T|$ 
        if  $i=t[n]$  then  $fg_1 \leftarrow 1$ ;
    if  $fg_1=1$  then continue;
    for  $j=w$  to  $l$ 
         $fg_2 \leftarrow 0$ ;
        for  $k=1$  to  $|S|$ 
            if  $v_2[j]=s[k]$  then
                 $fg_2 \leftarrow 1$ ; break;
        if  $fg_2=0$  then
             $v'[i] \leftarrow v_2[j]$ ;
             $w \leftarrow j+1$ ; break;
    output offspring ;
end
    
```

step 3 : produce offspring by filling unfixed position from parent 2



v_1 : parent chromosome 1 v_2 : parent chromosome 2
 l : length of chromosome v' : offspring chromosome
 N : total number of values in cycle
 $T=\{t[n]\}$, $n=1,2,\dots, N-1$: positions set of S in proto-child
 $S=\{s[k]\}$, $k=1,2,\dots, N-1$: proto-child genes value set in cycle
 $C=\{c[j]\}$, $j=1,2,\dots, N-1$: value set of cycle
 fg_1 : flag 1 fg_2 : flag 2
 w : working data
 $\text{cycle}(v_1, v_2)$: searching cycle between v_1 and v_2

3.2 Crossover Operators

6. Subtour Exchange Crossover

procedure : Subtour Exchange Crossover

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'_1, v'_2

begin

$S \leftarrow \emptyset, T \leftarrow \emptyset;$

// step 1: select subtours in parents

$s \leftarrow \text{random}[1:l-1];$

$t \leftarrow \text{random}[s+1:l];$

for $i=1$ **to** l

for $j=s$ **to** t

if $v_2[i]=v_1[j]$ **then**

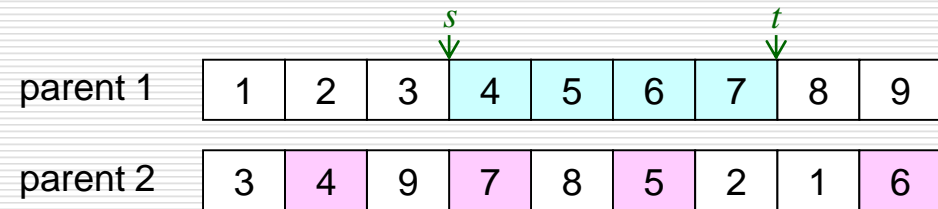
$S \leftarrow S \cup v_2[i];$

else

$v'_2[i] \leftarrow v_2[i],$

$T \leftarrow T \cup i;$

step 1 : select subtours in parents



v_1 : parent chromosome 1 v_2 : parent chromosome 2
 l : length of chromosome v'_1 : offspring chromosome 1
 v'_2 : offspring chromosome 2
 N : total number of values in subtour
 $S = \{s[i]\}, i=1,2,\dots, N$: value set of subtour
 $T = \{t[j]\}, j=1,2,\dots, N$: positions set of S
 s : start position of substring in v_1 t : end position of substring in v_1
 fg : flag

3.2 Crossover Operators

// step 2: exchange subtours

$v'_1 \leftarrow v_1[1:s-1] \text{ // } S \text{ // } v_1[t+1:l];$

$k \leftarrow s;$

for $i=1$ **to** l

$fg \leftarrow 0;$

for $j=1$ **to** $|T|$

if $i=t[j]$ **then** $fg \leftarrow 1;$ **break;**

if $fg=1$ **then** **continue;**

$v'_2[i] \leftarrow v_1[k];$

$k \leftarrow k+1;$

output offspring $v'_1, v'_2;$

end

step 2 : exchange subtours

offspring 1	1	2	3	4	7	5	6	8	9
offspring 2	3	4	9	5	8	6	2	1	7

v_1 : parent chromosome 1

v_2 : parent chromosome 2

l : length of chromosome

v'_1 : offspring chromosome 1

v'_2 : offspring chromosome 2

N : total number of values in subtour

$S = \{s[j]\}, j=1,2,\dots, N$: value set of subtour

$T = \{t[j]\}, j=1,2,\dots, N$: positions set of S

s : start position of substring in v_1 t : end position of substring v_1

fg : flag

3.3 Mutation Operators

1. Inversion Mutation

procedure : Inversion Mutation

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'

begin

// step 1: select subtour at random

$s \leftarrow \text{random}[1:l-1]$;

$t \leftarrow \text{random}[s+1:l]$;

// step 2: produce offspring by

copying inverse string of

substring

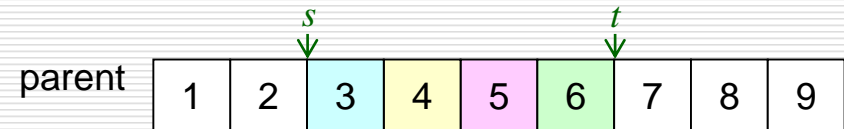
$S \leftarrow \text{invert}(v[s:t])$;

$v' \leftarrow v[1:s-1] // S // v[t+1:l]$;

output offspring v' ;

end

step 1: select subtour at random



step 2 : produce offspring by copying inverse string of substring



v : parent chromosome

v' : offspring chromosome

t : end position of substring

$\text{invert}(string)$: inversely changing order of $string$

l : length of chromosome

s : start position of substring

S : inverse string of substring

3.3 Mutation Operators

2. Insertion Mutation

procedure : Insertion Mutation

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'

begin

// step 1 : select a position in
parent 1 at random

$i \leftarrow \text{random}[1:l]$;

// step 2: insert selected value in
randomly selected
position parent 2

$j \leftarrow \text{random}[1:l-1]$;

$W \leftarrow v[1:i-1] // v[i+1:l]$;

$v' \leftarrow W[1:j-1] // v[i] // W[j:l-1]$;

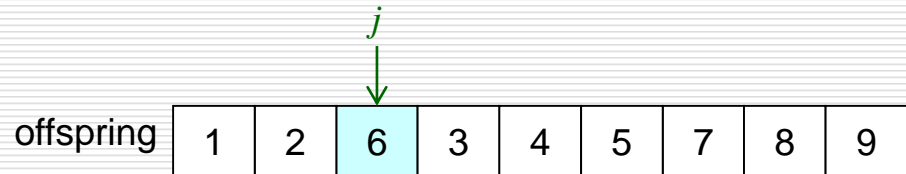
output offspring v' ;

end

step 1 : select a position in parent 1 at random



step 2: insert selected value in randomly
selected position of parent 2



v : parent chromosome
 v' : offspring chromosome
 j : selected position in parent 2

l : length of chromosome
 i : selected position in parent 1
 W : working data set

3.3 Mutation Operators

3. Displacement Mutation

procedure : Displacement Mutation

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'

begin

// step 1: select subtour

$s \leftarrow \text{random}[1:l-1]$;

$t \leftarrow \text{random}[s+1:l]$;

// step 2: insert subtour in a
random position

$n \leftarrow t - (s - 1)$;

$i \leftarrow \text{random}[1:l-n]$;

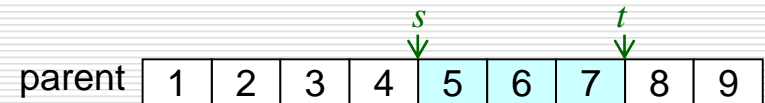
$W \leftarrow v[1:s-1] \text{ // } v[t+1:l]$;

$v' \leftarrow W[1:i-1] \text{ // } v[s:t] \text{ // } W[i:l-n]$;

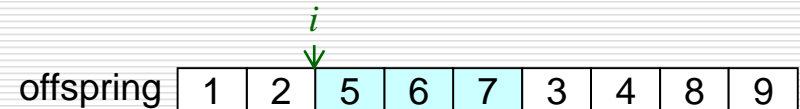
output offspring v' ;

end

step 1 : select subtour



step 2 : insert subtour in a random position



v : parent chromosome l : length of chromosome
 v' : offspring chromosome s : start position of substring
 t : end position of substring n : length of subtour
 i : insert position
 W : working data set

3.3 Mutation Operators

4. Swap Mutation

procedure : Swap Mutation

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'

begin

// step 1: select two position at random

$i \leftarrow \text{random}[1:l-1]$;

$j \leftarrow \text{random}[i+1:l]$;

// step 2: produce offspring by swapping

selected positions

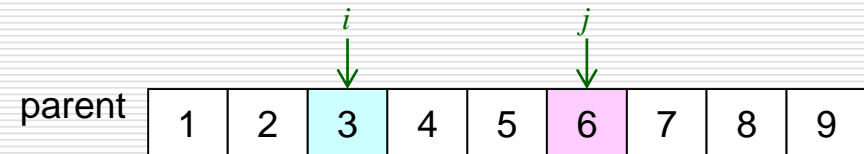
$v' \leftarrow v[1:i-1] \parallel v[j] \parallel v[i+1:j-1] \parallel v[i] \parallel v[j+1:]$

l]; v'

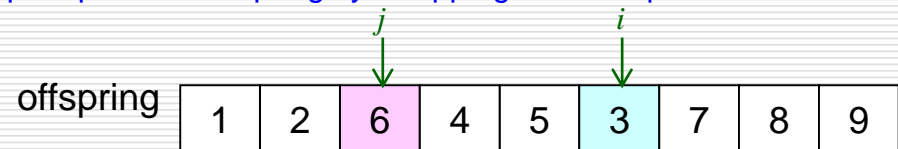
output offspring ;

end

step 1: select two position at random



step 2 : produce offspring by swapping selected positions



v : parent chromosome

v' : offspring chromosome

j : selected position

l : length of chromosome

i : selected position

3.3 Mutation Operators

5. Heuristic Mutation

procedure : Heuristic Mutation

input : chromosome v_1, v_2 ,
length of chromosome l

output : offspring v'

begin

$P \leftarrow \emptyset$;

// step 1: select positions and
produce neighbors

for $i=1$ **to** m {

$r \leftarrow \text{random}[1:l]$;

$P \leftarrow P \cup \text{nb}(v[r])$;

}

// step 2: produce offspring by
evaluating neighbors

$w \leftarrow F(p[1])$;

for $i=2$ **to** $|P|$

if $w > F(p[i])$ **then**

$w \leftarrow F(p[i])$, $n \leftarrow i$;

$v' \leftarrow p[n]$;

output offspring v' ;

end;

step 1 : select positions and produce neighbors

parent	1	2	3	4	5	6	7	8	9	$F(p[i])$
			$r \downarrow$			$r \downarrow$		$r \downarrow$		32
proto-child 1	1	2	3	4	5	8	7	6	9	27
proto-child 2	1	2	8	4	5	3	7	6	9	54
proto-child 3	1	2	8	4	5	6	7	3	9	45
proto-child 4	1	2	6	4	5	8	7	3	9	23
proto-child 5	1	2	6	4	5	3	7	8	9	56

step 2 : produce offspring by evaluating neighbors

offspring	1	2	6	4	5	8	7	3	9
-----------	---	---	---	---	---	---	---	---	---

v : parent chromosome l : length of chromosome
 v' : offspring chromosome m : total number of selected positions
 r : selected position N : total number of neighbor chromosomes
 w : working data
 n : position of chromosome with best fitness value in P
 $P\{p[i]\}, i=1,2,\dots,N$: neighbor chromosome set
 $\text{nb}(v[r])$: searching neighbors of r th gene
 $F(p[i])$: fitness value of $p[i]$

3.4 Overall Algorithm

□ GA procedure for Traveling Salesperson Problem

procedure: GA for Traveling Salesperson Problem (TSP)

Input: TSP data set, GA parameters

output: best tour route

begin

$t \leftarrow 0$;

initialize $P(t)$ by permutation encoding or random keys encoding;

fitness $eval(P)$ by permutation decoding or random keys decoding;

while (not termination condition) **do**

 crossover $P(t)$ to yield $C(t)$ by partial-mapped crossover;

 mutation $P(t)$ to yield $C(t)$ by swap mutation;

 fitness $eval(C)$ by permutation decoding or random keys decoding;

 select $P(t+1)$ from $P(t)$ and $C(t)$;

$t \leftarrow t+1$;

end

output best tour route;

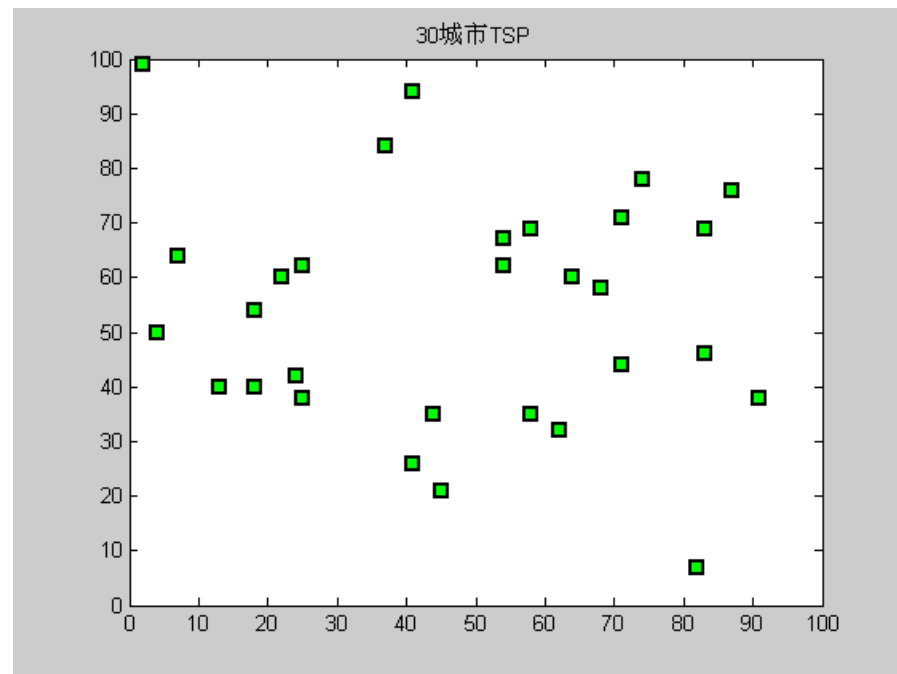
end

3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

◆ TSP Benchmark 问题

41 94;37 84;54 67;25 62;
7 64;2 99;68 58;71 44;54
62;83 69;64 60;18 54;22
60;83 46;91 38;25 38;24
42;58 69;71 71;74 78;87
76;18 40;13 40;82 7;62 32;
58 35;45 21;41 26;44 35;4 50



3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

- ◆ 初始温度的计算

```
for i=1:100
```

```
    route=randperm(CityNum);
```

```
    fval0(i)=CalDist(dislist,route);
```

```
end
```

```
t0=-(max(fval0)-min(fval0))/log(0.9);
```

3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

◆ 状态产生函数的设计

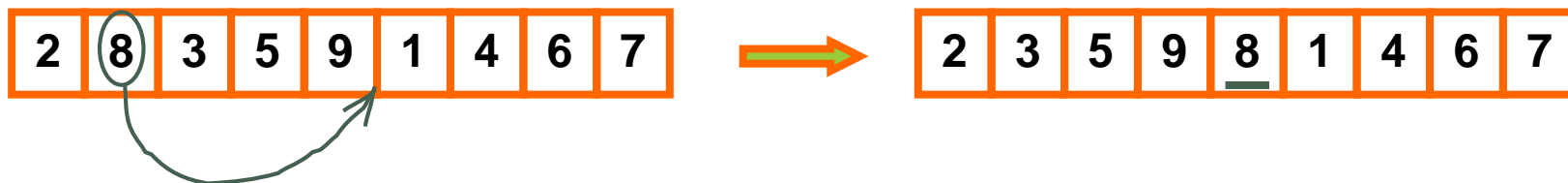
(1) 互换操作, 随机交换两个城市的顺序;



(2) 逆序操作, 两个随机位置间的城市逆序;



(3) 插入操作, 随机选择某点插入某随机位置。



3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

- ◆ **参数设定**

截止温度 $tf=0.01$;

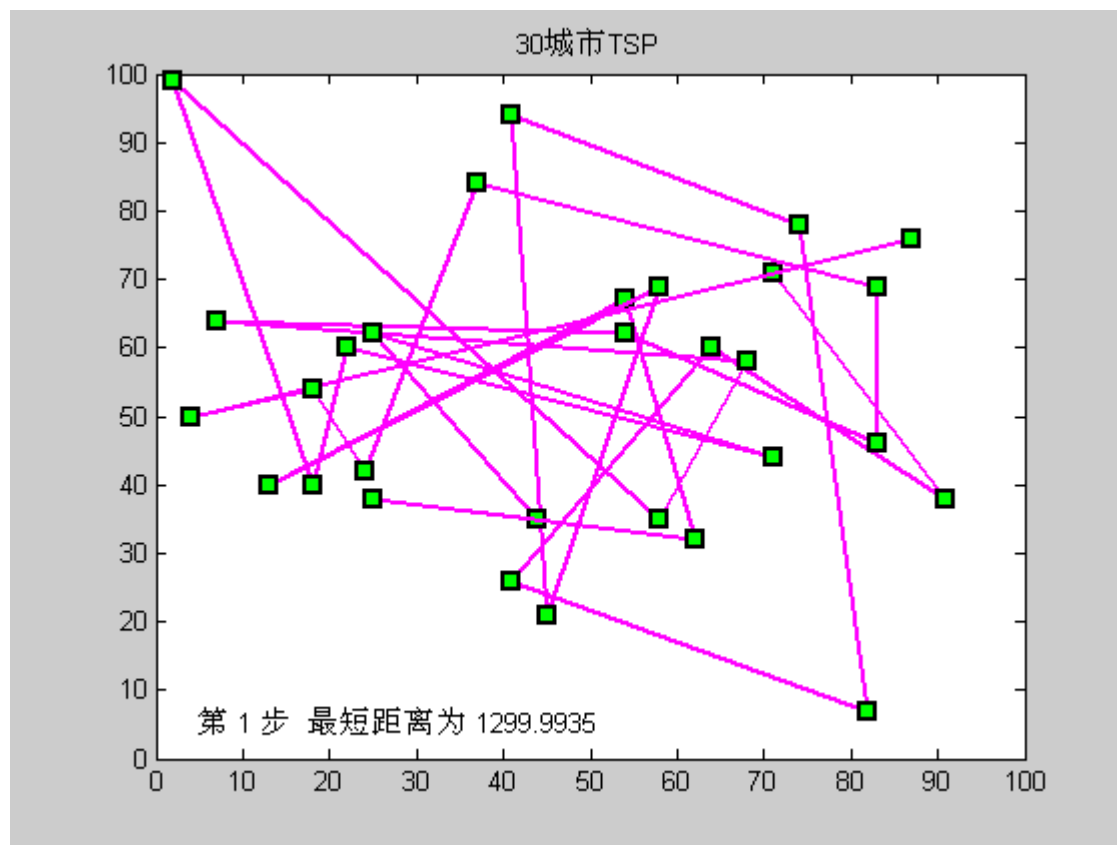
退温系数 $\alpha=0.90$;

内循环次数 $L=200*CityNum$;

3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

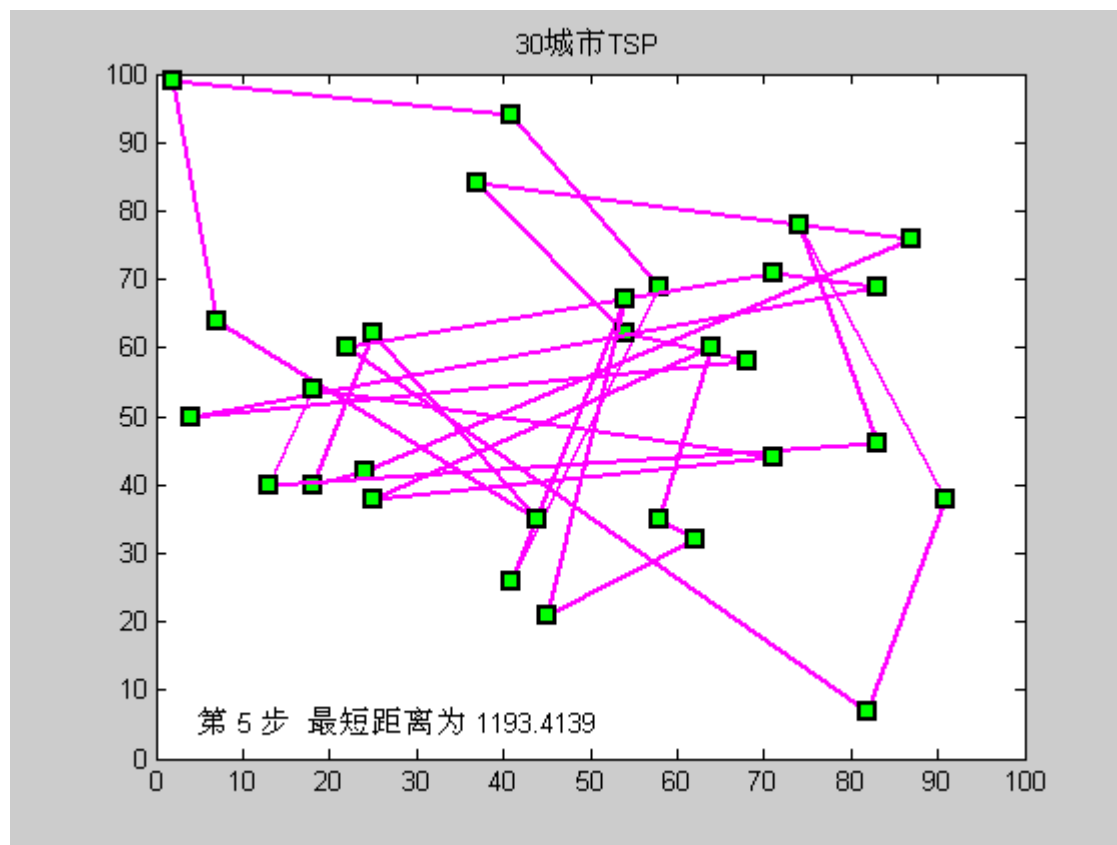
◆ 运行过程



3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

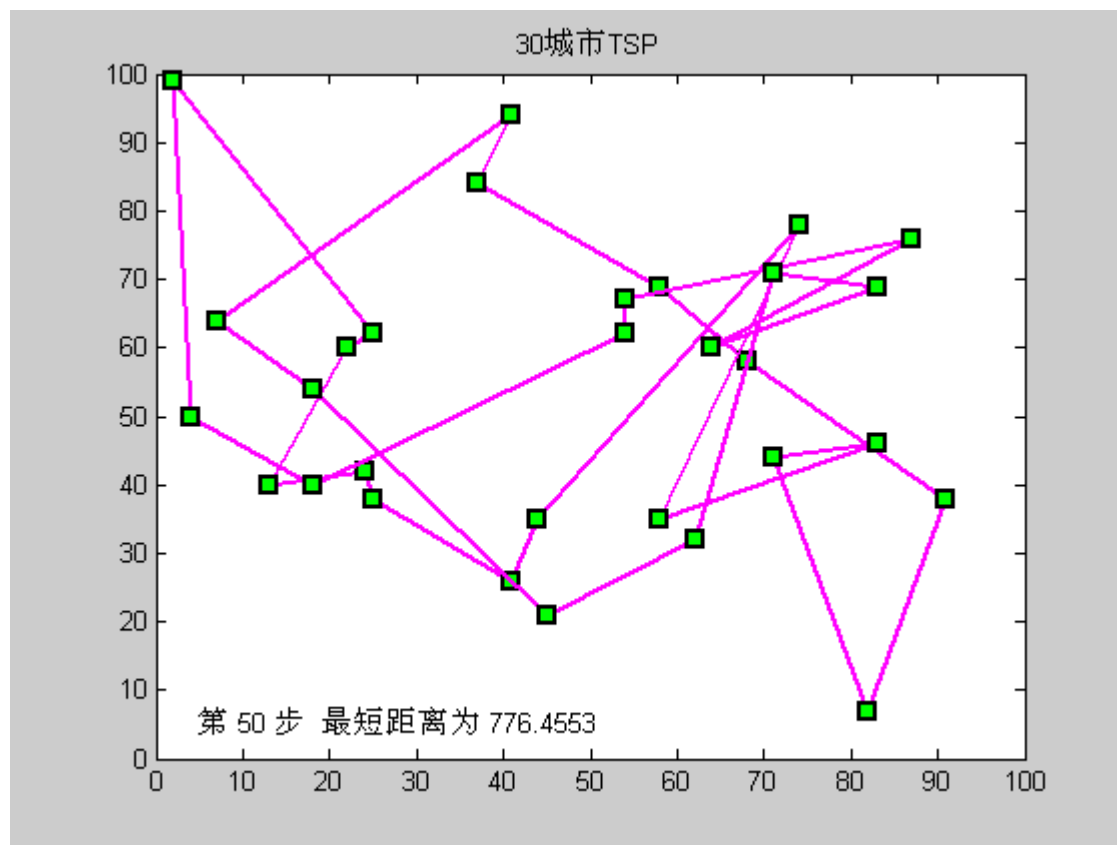
◆ 运行过程



3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

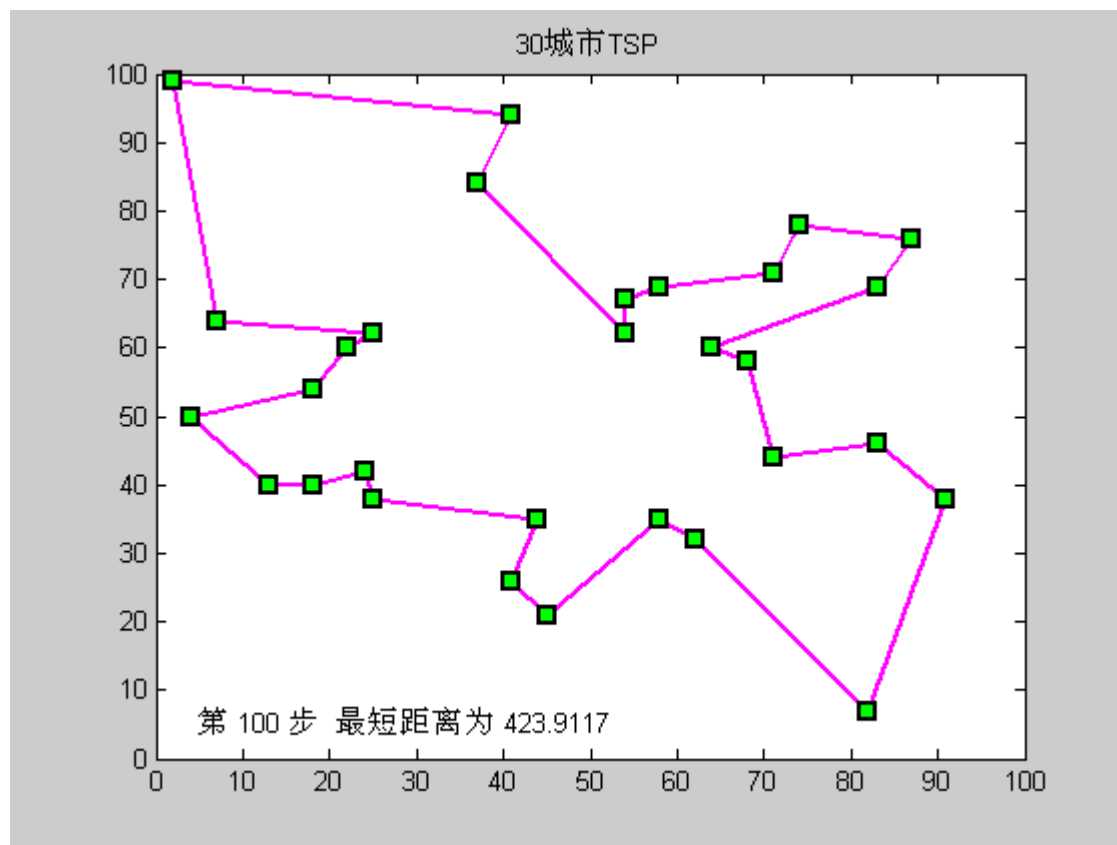
◆ 运行过程



3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

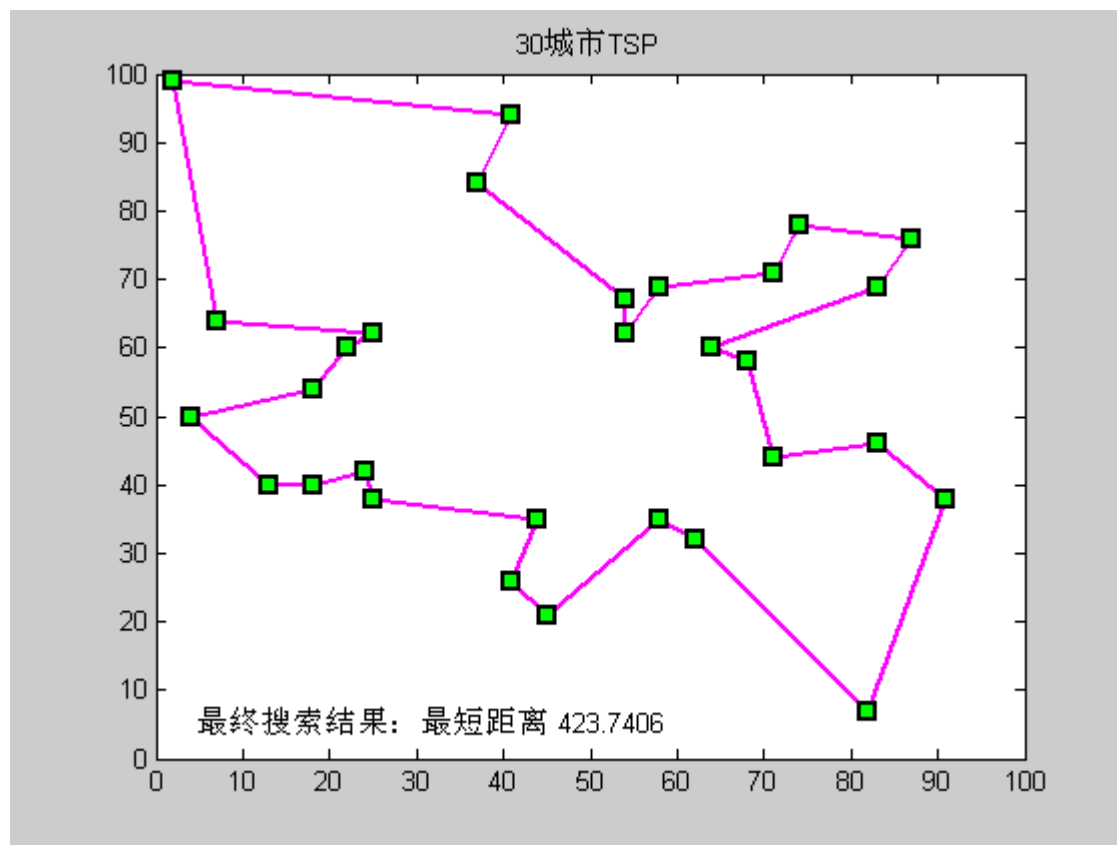
◆ 运行过程



3. Traveling Salesman Problem

30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

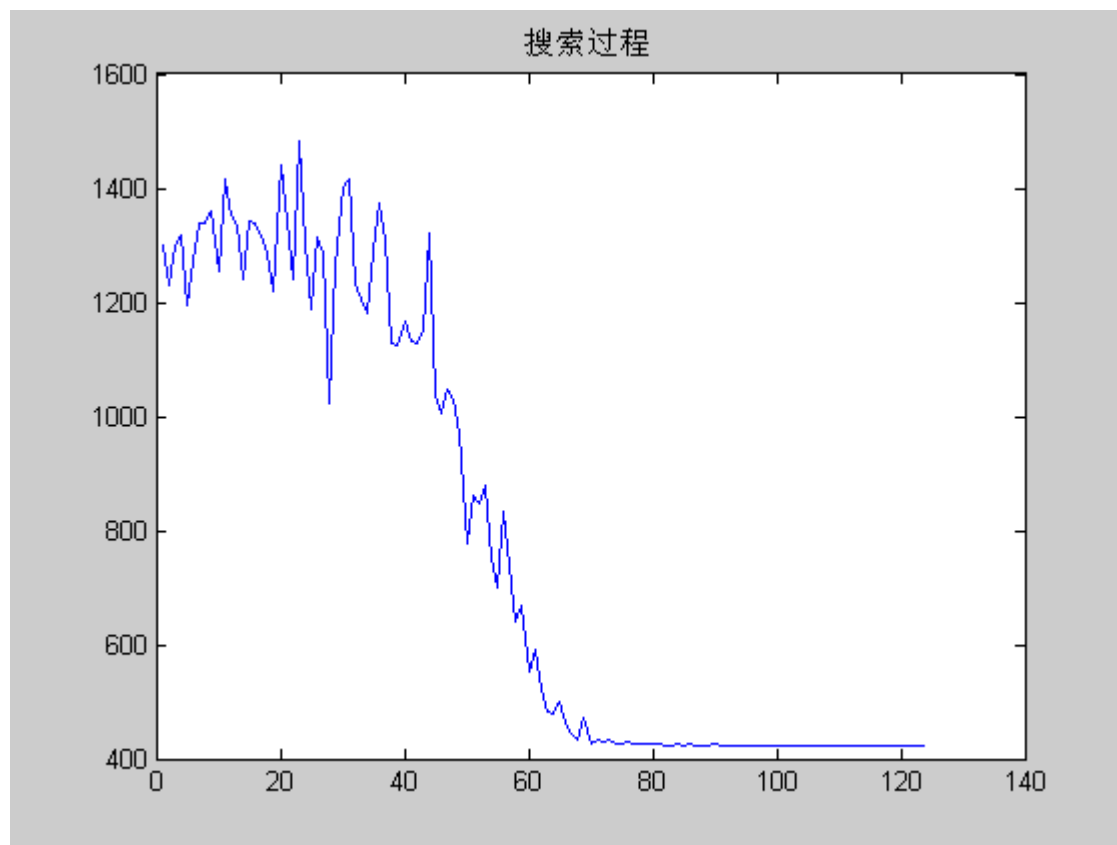
◆ 运行过程



3. Traveling Salesman Problem

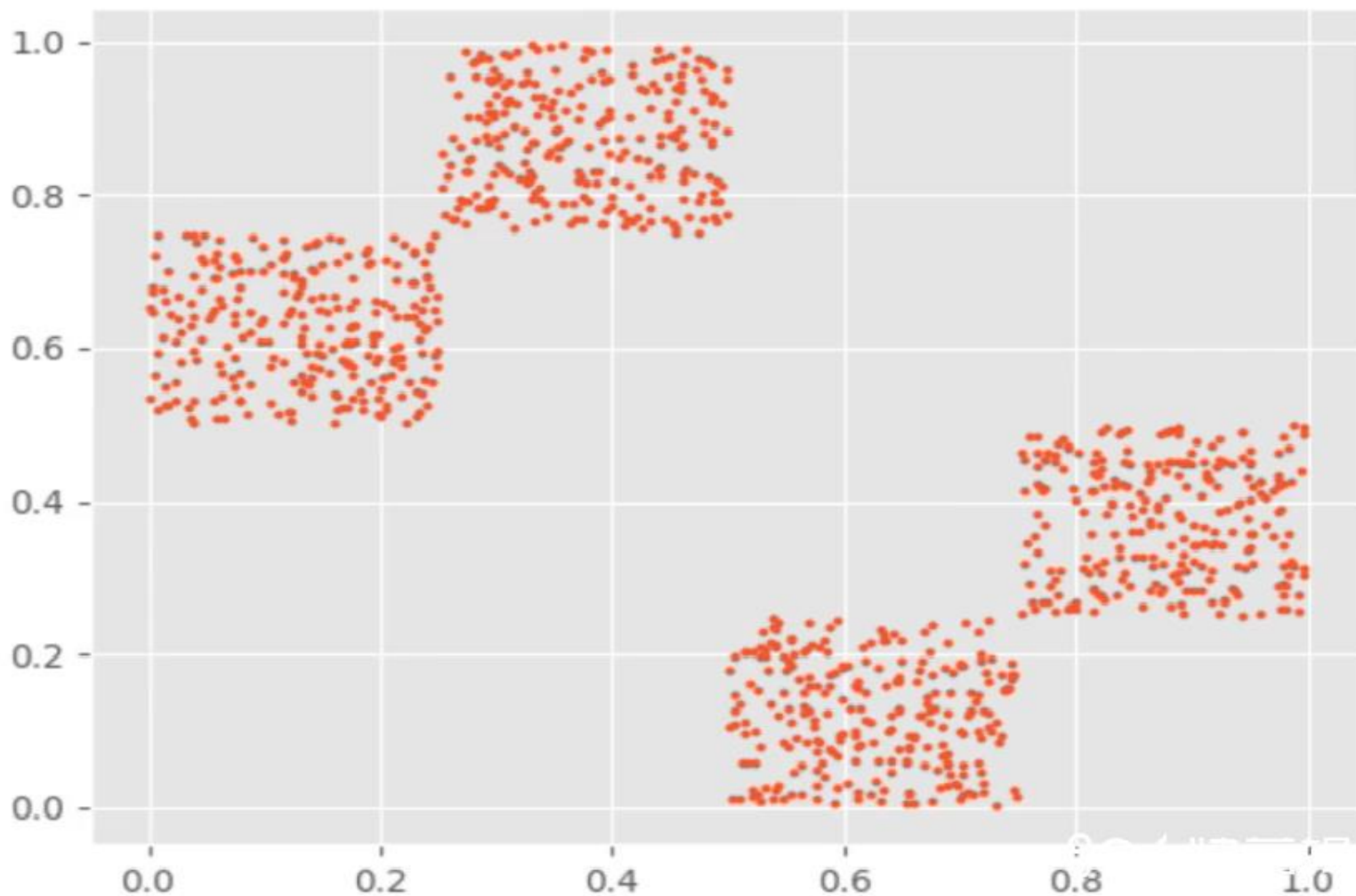
30城市TSP问题 ($d^*=423.741$ by D B Fogel) 模拟退火

◆ 运行结果



不同的搜索算法比较：最短路径问题案例

不会玩游戏的程序猿 bilibili





Thanks

Thanks the support from Prof. Wang and the authors of the textbook, for the refer to their slides.