Dijkstra 算法与遗传算法研究心得

1951976 李林飞

日期：2021 年 3 月 27 日

摘　　要

本文基于阅读论文《A Note on Two Problems in Connexion with Graphs》、《遗传算法在优化问题中的应用研究进展》和《改进自适应遗传算法在函数优化中的应用研究》后的感想，具体研究了 Dijkstra 算法和遗传算法的核心思想与效率分析，并应用 Dijkstra 算法解决最短路问题、应用遗传算法解决 TSP 问题。

关键词： 遗传算法 (Genetic Algorithm), Dijkstra Algorithm, TSP 问题, 最短路径问题

目录

1 算法背景介绍

1.1 Dijkstra 算法

Dijkstra 算法是一种用于查找图形中节点之间最短路径的算法，该路径可能表示例如道路网络。它是由计算机科学家 Edsger W. Dijkstra 于 1956 年构思的。该算法存在多种变体。Dijkstra 的原始算法找到了两个给定节点之间的最短路径，但更常见的变体将单个节点固定为“源”节点，并找到从源到图中所有其他节点的最短路径，从而产生最短路径树。

1.2 GA 算法

在计算机科学和运筹学中，遗传算法（GA）是受自然选择过程启发的一种元启发法，它属于较大类的进化算法（EA）。遗传算法通常用于依靠诸如突变，交叉和选择等受到生物学启发的运算符来生成用于优化和搜索问题的高质量解决方案。

2 Dijkstra 算法

2.1 Dijkstra 算法核心思想

在通过分支连接的 n 个结点中，给出每个分支的长度都是已知的。在假设任意两个结点之间都至少存在一条路径的条件下，我们考虑以下两个问题。

问题 1：构造最小生成树

构造最小生成树，即使 n 个节点之间的总长度最小的树。树是一种图形，每两个节点之间只有一条路径。为了方便问题讨论，先将分支分为三类：

I. 明确分配给正在建造的树的分支（它们会形成一个子树）;

II. 将要添加到集合 I 的下一个分支的分支;

III. 其余分支（被拒绝或尚未考虑）。

同样，节点又分为两组：

A. 由集合 I 的分支连接的节点；

B. 其余节点（集合 II 中只通过一个分支连接的结点）。

构建过程从集合 A 中任意选一个结点作为其唯一的成员开始，然后将与该结点相连的所有结点加入集合 II 中。同时，将集合 I 设为空，接着重复执行以下步骤：

$Step1$: 将集合 II 中最短的分支添加到集合 I 中，然后将其从集合 II 删除。同时将从集合 B 中加入一个新的结点到集合 A 中。

$Step2$: 考虑到与该结点通过分支相连的结点还在集合 B 中。如果与之相连的分支长度比在集合 II 相应的分支要长，则将其拒绝；如果更短，则它将取代在集合 II 中的分支，并将后者从集合 II 中删除。

然后，我们返回到 $Step1$ 中并重复该过程，直到集 II 和 B 为空。此时集合 I 中的分支即是我们寻找的最小生成树。

这种方法比 kruskal 算法更好，因为 Kruskal 算法首先需要将所有分支按长度进行排序，其次数大概为 $\frac{1}{2}n(n-1)$。即使分支长度是可计算的函数坐标，但实现的过程中也需要同时存储所有分支的信息。而 Dijkstra 算法最多只需要存储 n 个分支的信息，即在集合 I、II 以及在 $Step2$ 中被考虑的分支。

问题 2：任意两个结点之间的最短路径

假设要找到两个给定结点 P 和 Q 之间的最小总长度的路径。

基于这样一个事实：如果 R 是从 P 到 Q 的最短路径上的一个结点，则此路径也是 P 到 R 的最短路径。因此，从 P 点到 Q 点的最短路径等价于从 P 点到其他点的最短路径，直到该路径延伸到 Q 点。

为了方便解决问题，我们先将所有结点分为三组：

A. 到 P 点的最短路径是已知的结点。然后按照最短路径长度增加的顺序将结点加入到该集合中；

B. 要添加到集合 A 的下一个节点的结点，即该集合包括了至少与集合 A 中的结点相连一次，但并不在集合中的所有结点；

C. 其余的结点。

同样，分支也分为三组：

I. 在结点 P 到集合 A 中各结点的最短路径中出现的分支；

II. 下一个将加入到集合 I 中的分支选择。该集合中的分支与集合 B 中每一个结点相连的有且只有一个；

III. 其余分支（被拒绝或尚未考虑）。

实现过程先将所有结点加入到集合 C 中，所有分支加入到集合 III 中。将 C 中的 P 结点转移到集合 A 中，然后开始重复以下步骤：

$Step1$: 考虑到与刚转移到集合 A 中的结点相连的所有分支 r 在集合 B 或集合 C 中有结点 R 与之相连。如果结点 R 属于集合 B 中，我们判断通过分支 r 从结点 P 到 R 之间的路径长度是否比在集合 II 中的最短路径短。如果不是，则拒绝分支 r；否则用分支 r 取代集合 II 中的最短路径。如果结点 R 属于集合 C 中，则将其添加到集合 B 中，同时将分支 r 加入到集合 II 中。

$Step2$: 如果我们只选取集合 I 或 II 中的分支，则每个在集合 B 中的结点都能通过唯一一条路径与结点 P 相连。因此，集合 B 中的结点到结点 P 都存在一个有限的距离：将具有最短距离的结点从集合 B 中转移到集合 A 中，并将与之相连的分支从集合 II 中转移到集合 I 中。

然后返回 $Step1$ 并重复该过程直到将结点 Q 加入到集合 A 中。此时，集合 A 中的结点构成了从 P 点到 Q 点的最短路径。

集合 B 中的每个节点只能以一种方式连接到节点 P 如果我们将自己限制为 I 组的分支和 II 组的分支。在这个意义上说集合 B 中的每个节点到节点 P 的距离：具有最小距离的节点从 P 转移到集合 B 到集合 A，相应的分支被转移从第 II 组到第 I 组。然后返回到步骤并重复该过程直到将节点 Q 转移到集合 A。然后找到了解决方案。

在该算法中，无论分支的数量如何，我们不需要同时存储所有分支的数据，而只需要存储集合 I 和 II 中的分支，并且此数字始终小于 n。

2.2 Dijkstra 算法伪代码

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
9      dist[source] ← 0
10
11     while Q is not empty:
12         u ← vertex in Q with min dist[u]
13
14         remove u from Q
15
16         for each neighbor v of u:           // only v that are still in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] ← alt
20                 prev[v] ← u
21
22     return dist[], prev[]
```

图 1: Dijkstra 算法伪代码 [1]

2.3 基于 Dijkstra 算法解决最短路径问题

如图所示，求解点 a 到点 b 的最短路径。从 a 点开始，先找到与 a 点相连的路径最短的结点，即 2 号点。将点 a 和 2 号点放在同一个集合中，然后寻找与这两点相连的路径最短的点，即 3 号点，加入集合中。依次类推，当 b 点也加入到集合中时，该集合中结点连成的路径即为点 a 到点 b 的最短路径。具体实现代码详见附录。

2.4 Dijkstra 算法分析

Dijkstra 的算法使用一种数据结构来存储和查询从一开始就按距离排序的部分解决方案。在原始算法使用了最小优先级队列，其时间复杂度为 $\Theta(|V|+|E|log|V|)$（其中 $|V|$ 为图结构结点数，$|E|$ 为边数）。如果用数组存储图结构，则时间复杂度为 $\Theta(|V|^2)$。因此，降低 Dijkstra 算法的复杂度的方法就是降低排序过程中的时间复杂度。如果使用 Fibonacci Heap，则时间复杂度可降为 $\Theta(|E|+|V|log|V|)$。

3 遗传算法 (Genetic Algorithm)

3.1 GA 算法核心思想

遗传算法是一类借鉴生物学中的进化规律（适者生存，优胜劣汰遗传机制）演化而来的随机优化搜索方法。其特点是直接对结构对象进行操作，不存在求导和函数连续性的限定；具有

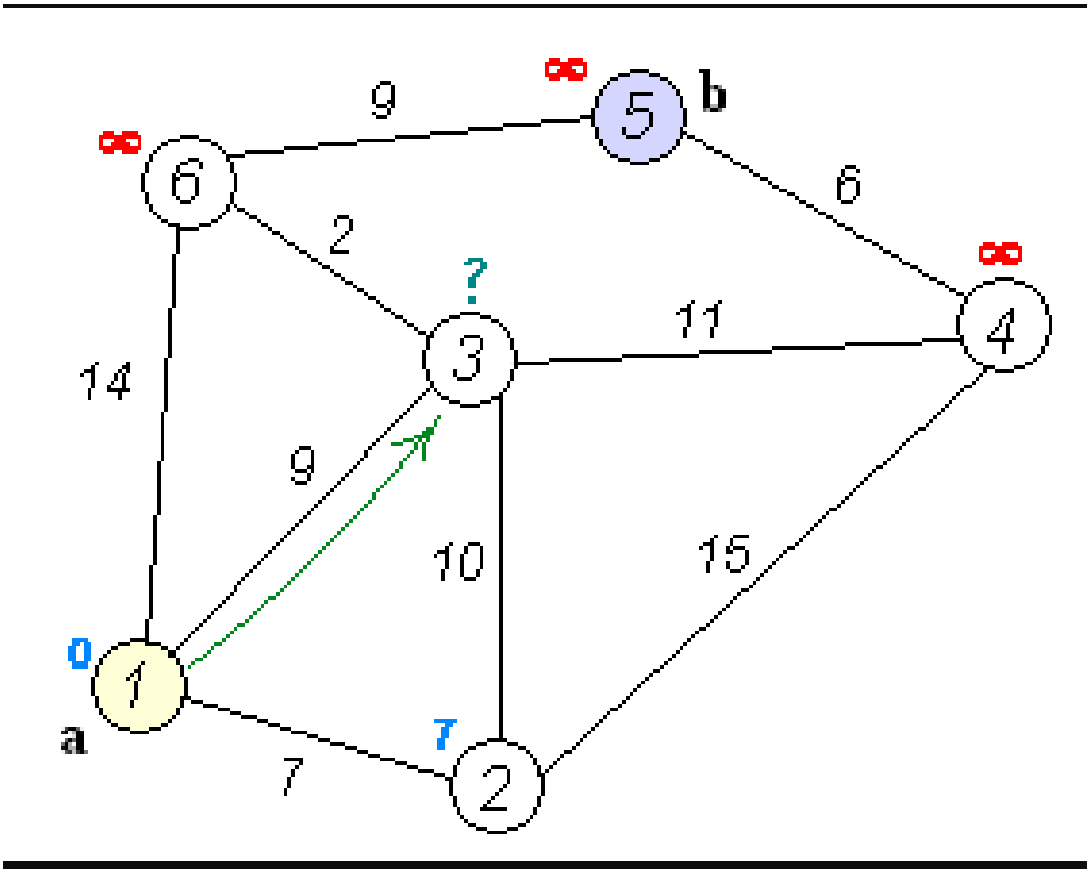

图 2: 图结构示例

内在的隐并行性和更好的全局寻优能力；采用概率化的寻优方法，能自动获取和指导优化的搜索空间，自适应地调整搜索方向，不许呀确定的规则。随着算法的优化与普及，其应用已经衍生到机器学习、神经网络、模糊推理、并行计算等多个方面。

遗传算法的基本步骤如下：

初始化群体 (Initial population)

因为遗传算法不能直接处理问题空间的参数，必须把他们转换成遗传空间的由基因按一定结构组成的染色体或个体。这个转化过程叫做编码。即把所需要选择的特征进行编号，每一个特征就是一个基因，一个解就是一串基因的组合。为了减少组合数量，在图像中进行分块，然后再把每一块看成一个基因进行组合优化的计算。每个解的基因数量是通过实验确定的。

评估编码策略常用三个性质：

1）完备性 (Completeness)：问题空间中所有点 (候选解) 都能作为 GA 空间中的点 (染色体) 表现；

2）健全性 (Soundness):GA 空间中染色体能对应所有问题空间中的候选解；

3）非冗余性 (Nonredundancy): 染色体和候选解一一对应。

目前常用的几种编码技术有二进制编码、浮点数编码、字符编码等。经过编码后，可以进行初始群体的生成。

随机产生 N 个初始串结构数据，每个串结构数据称为一个个体。N 个个体构成一个群体。然

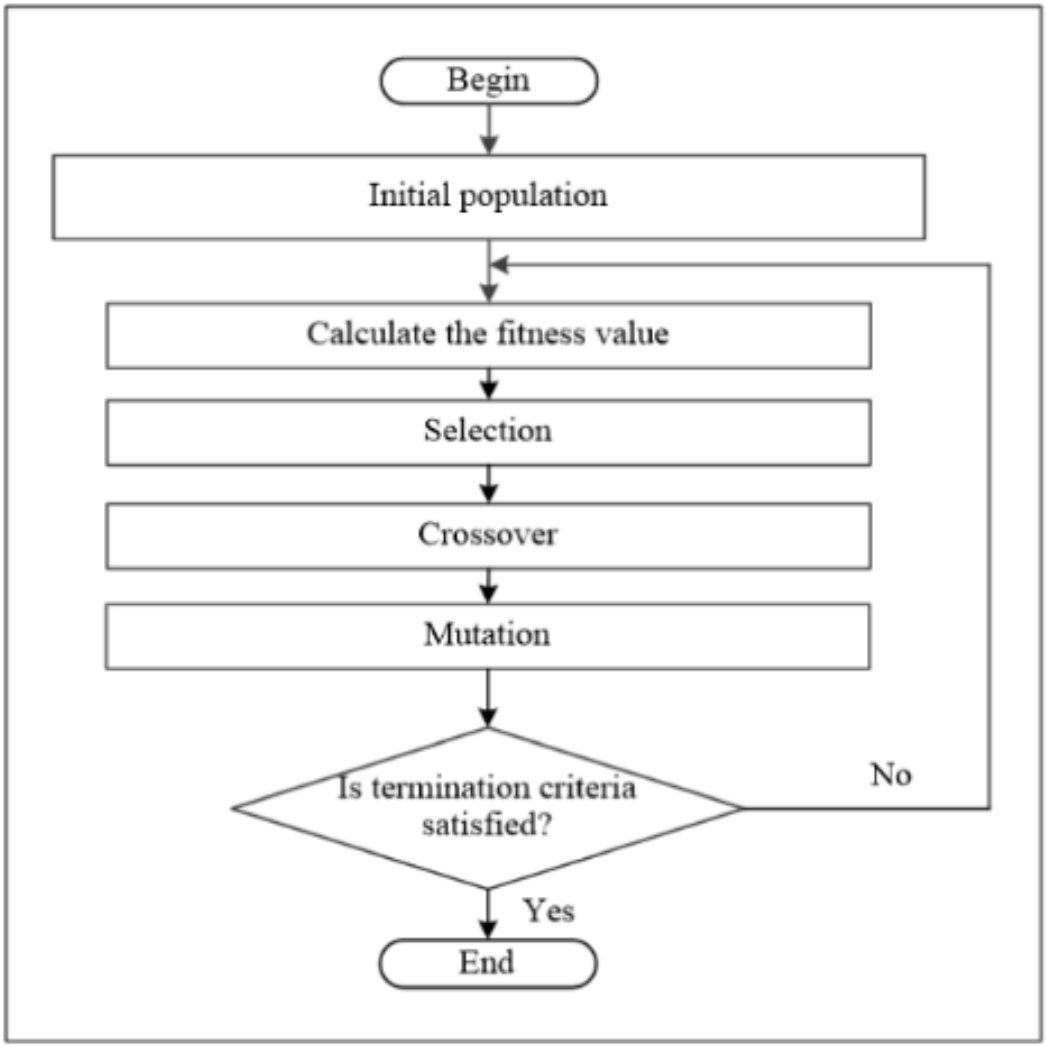

图 3: GA 算法流程图

后以这 N 个初始串结构数据作为初始点开始迭代，其中参数 N 是根据问题的规模确定的。进化论中的适应度，是表示一个个体对环境的适应能力，也表示该个体繁殖后代的能力。在算法中与之对应的是适应度函数，也叫评价函数，是用来判断群体中的个体的优劣程度的指标，它是根据所求问题的目标函数来进行评估的。遗传算法中初始群体中的个体是随机产生的。

一般来讲，初始群体的设定可采取如下的策略：

1）根据问题的固有知识，设法把握最优解所占空间在整个问题中的分布范围，然后，在此分布范围内设定初始群体；

2）先随机生成一定数目的个体，然后从中挑出最好的个体加到初始群体中。这种过程不断迭代，直到初始群体中的个数达到了预先确定的规模。

杂交 (Crossover)

杂交操作是遗传算法中最主要的遗传操作。由交换概率挑选的没两个父代通过将相异的部分基因进行交换，从而产生新的个体，新个体组合了其父辈个体的特征。杂交体现了信息交换的思想。

适应度值评估检测 (Calculate the fitness value)

计算交换产生的新个体的适应度。适应度是用来度量种群中个体优劣的指标，这里的适应度就是特征组合的判据的值。这个判据的选取是遗传算法的关键。

遗传算法在搜索进化过程中一般不需要其他外部信息，仅用评估函数来评估个体或解的优劣，并作为以后遗传操作的依据。由于遗传算法中，适应度函数要比较排序并在此基础上计算选择概率，所以适应度函数的值要取正值。由此可见，在不少场合，将目标函数映射成求最大值形式且函数值非负的适应度函数是必要的。

适应度函数的设计主要需要满足以下条件：

1）单值、连续、非负、最大化；

2）合理、一致性；

3）计算量小；

4）通用性强。

选择 (Selection)

选择的目的是为了从交换后的群体中选出优良的个体，使他们有机会作为父代为下一代繁衍子孙。进行选择的原则是适应性强的个体为下一代贡献的概率大，体现了达尔文的适者生存原则。

变异 (Mulation)

变异首先在群体中随机选择一定数量的个体，对于选中的个体以一定的概率随机地改变串结构数据中的某个基因的值。同生物界一样，遗传算法中变异发生的概率很低，通常取值在 $0.001 - 0.01$ 之间。变异为新个体的产生提供了机会。

中止 (End)

中止的条件一般有三种情况：

1）给定一个最大遗传代数，算法迭代到最大代数时为止；

2）给定问题一个下界的计算方法，当进化中达到要求的偏差时，算法终止。

3）当监控得到的算法再进化已无法改进解的性能时停止。

3.2 GA 算法伪代码

3.3 基于 GA 算法解决 TSP 问题

旅行商问题 (Traveling Salesman Problem, TSP) 是一个经典的 NP-hard 问题，求解该类问题一般有两个思路：一种是与问题特征相关的启发式搜索算法，主要有动态规划法、分支界定法等；另一种是独立于问题的智能优化算法，如模拟退火法、禁忌搜索法、蚁群算法、遗传算法和粒子群算法等。接下来，我们尝试用 GA 算法求解 TSP 问题。

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

图 4: GA 算法伪代码 [2]

TSP 模型化

为了方便解决 TSP 问题，首先我们将其模型化。给定 n 各城市各城市之间的距离，寻找一条遍历所有城市且每个城市只被访问一次的路径，并保证总路径距离最短。将其用数学语言表述如下：

设 $G = (V, E)$ 为赋权图，$V = 1, 2, ..., n$ 为顶点集，E 为边集，各顶点间距离为 C_{ij}，已知 $(C_{ij} > 0, i, j \in V)$, 并设

$$x_{ij} = \begin{cases} 1 & \text{在最优路径上} \\ 0 & \text{others} \end{cases}$$

则 TSP 问题可用如下数学模型 (1)-(5) 式联立表示：

$$minZ = \sum_{i \neq j} c_{ij} x_{ij} \tag{1}$$

$$\sum_{i \neq j} x_{ij} = 1 \quad j \in v \tag{2}$$

$$\sum_{j \neq i} x_{ij} = 1 \quad j \in v \tag{3}$$

$$\sum\nolimits_{i,j \in s} x_{ij} \leq |K| - 1 \quad k \in v \tag{4}$$

$$x_{ij} \in 0, 1 \quad i, j \in v \tag{5}$$

其中，K 是 V 的全部非空子集，$|K|$ 是集合 K 中包含图 G 的全部顶点的个数。

基于 GA 算法的 TSP 求解步骤

（1）种群初始化。个体编码方法有二进制编码和实数编码，在解决 TSP 问题过程中个体编码方法为实数编码。对于 TSP 问题，实数编码为 1-n 的实数的随机排列，初始化的参数有种群的个数 M、染色体基因个数 N（即城市的个数）、迭代次数 C、交叉概率 P_c 和变异概率 P_m。

（2）适应度函数。在 TSP 问题中，对于任意两个城市之间的距离 $D(i,j)$ 已知，每个染色体（即 n 个城市的随机排列）可计算出总距离，因此可将一个随机全排列的总距离的倒数作为适应度函数，即距离越短，适应度函数越好，满足 TSP 要求。

（3）选择操作。遗传算法选择操作有轮盘赌法、锦标赛法等多种方法，本文采用基于适应度比例的选择策略，即适应度越好的个体被选择的概率越大，同时在选择中保存适应度高的个体。

（4）交叉操作。遗传算法中交叉操作有多种方法。本文中对于个体，随机选择两个个体，在对应位置交换若干基因片段，同时保证每个个体依然是 1-n 的随机排列，防止进入局部收敛。

（5）变异操作。本文中对于变异操作，随机选取个体，同时随机选取个体的两个基因进行交换以实现变异操作。

TSP 算法实现

由于 matlab 中有自带的 TSP 问题中美国城市地图，故使用 matlab 实现，代码详见目录。接着我们讨论实现过程中三种遗传算子的选取。

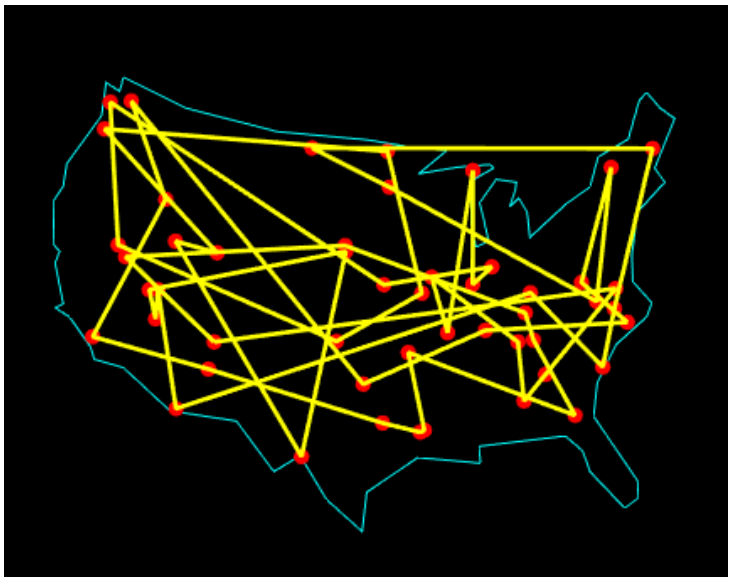

图 5: 美国 50 城市地图

选择 (Selection) 的目的是把优化的个体直接遗传到下一代或通过交叉配对产生新的个体再遗传给下一代。其中最简单的是轮盘赌选择法 (Roulete Wheel Selection)，在该方法中，各个体的选择概率和其适应度值成比例。设群体大小为 n，其中个体 i 的适应度为 f_i，则 i 被选择的概率为

$$P_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

显然，概率反应了个体 i 的适应度在整个群体的个体适应度总和中所占的比例。个体适应度越大，其被选择的概率就越高，反之亦然。计算出群体中各个个体的选择概率之后，为了选择交配个体，需要进行多轮选择。每一轮选择产生 $[0,1]$ 之间的均匀随机数，将该随机数作为选择指针来确定被选个体。个体被选后，可随机地组成交配对，以供后面交叉操作。

交叉 (Crossover) 是指把两个父代个体的部分结构加以替换重组而生成新个体的操作。其目的是提高算法的搜索能力。交叉算子根据交叉率将种群中的两个个体随机地交换某些基因，能够产生新的基因组，期望将有益基因组合在一起。目前最常用的交叉算子是单点交叉 (one-point crossover)。具体操作是，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交换，并生成两个新的个体。例子如下：假设为有两个个体 AB

$$A : 1001!111$$

$$B : 0011!000$$

从符号“!”后面开始交换，则产生两个新的个体 A', B'

$$A' : 1001000$$

$$B' : 0011111$$

变异算子 (Mulation) 的基本内容是对群体中的个体串的某些基因座上的基因值作变动。其目的为：一是使算法具有局部搜索的能力。当遗传算法通过交叉算子已接近最优解邻域时，利用变异算子可以加速向最优解收敛；二是维持种群多样性，以防止出现未成熟收敛现象。此时收敛概率应取最大值。

求解结果如下图所示：

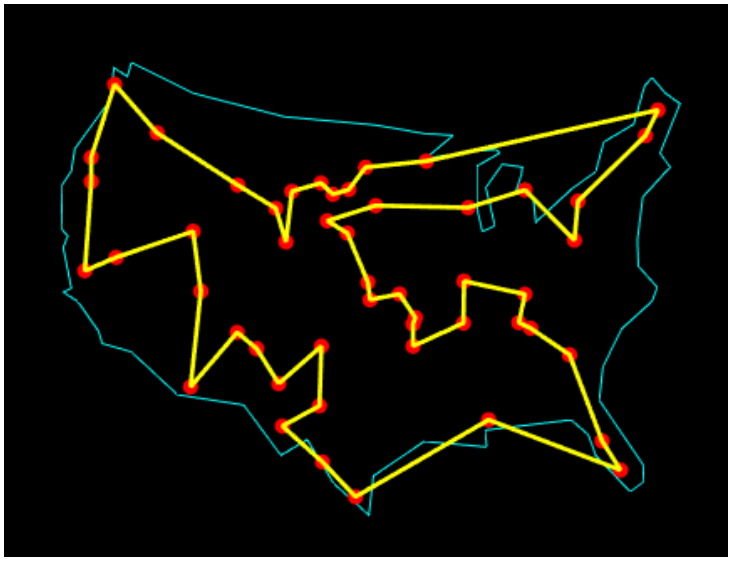

图 6: 美国 50 城市 TSP 求解结果图

4 论文学习感想

关于 Dijkstra 算法的论文《A Note on Two Problems in Connexion with Graphs》，我被作者简单而深邃的思想所惊讶。首先，相比于目前大量论文，该论文篇幅短小，但里面的算法思想却通

俗易懂。这对科研工作时一种启示：想法是最重要的！一个好的想法并不需要太多的文字来装饰。看五十多年前的论文，确实能体会到当时科研人所秉持的一些精神。他们的科研从不追求多，而是精。

对于遗传算法的学习，由于其核心思想来源于生物学进化论，所以许多专有名词会让人不明所以，尤其在看一篇关于遗传算法应用研究的文章，理解起来是很困难的。对于这两篇论文，我只是从中了解了遗传算法的大概内容，其具体细节以及实现是翻看一些文献后理解的。

看论文，理性的分析固然重要，但对我而言，其带来的感性上的冲击更大。曾经在学习 DFS 和 BFS 算法时，我就设想过通过概率让两个算法在同一个巨型网络中实现搜索，其思想是源于一个人在一个巨型迷宫里探索时，他采取的措施是“试探”，即是一个广搜加深搜的过程。虽然这样的想法可能是荒诞的。但从 Dijkstra 算法以及遗传算法中可以看出，一个算法往往是来源于自然界中某种事物面临某些问题时采取的自然回应。就像蜗牛壳上欧几里得螺旋线的形成是因为把重量更均匀的摊分。模拟退火算法也是受固体降低过程的启发。基于这些伟大思想的启发，我对自然界中的“某种力量”产生敬畏之情。什么是最优的？似乎自然界在进化中选择的结果就是一个无限趋于最优的过程。

参考文献

[1] Goodrich and Tamassia; Cormen, Leiserson, Rivest, and Stein.

[2] http://aima.cs.berkeley.edu/algorithms.pdf

附录

GA 算法求解 TSP 问题的 matlab 代码

```
clear all;

clc;

chk1='Please␣enter␣the␣number␣of␣cities␣(N)␣:';
N=input(chk1);

%% Function to evaluate the distance between two points

Rd=@(x1,y1,x2,y2)(sqrt(((x2-x1)^2)+((y2-y1)^2)));

%% generation of the N coordinates

XY=100*rand(N,2);
XY1=[];
for i=1:N
XY1=[XY1;i XY(i,:)];
end

%% creating random set of 0.2*N parents to choose for mutation

Np=transpose(XY1(2:end,1));  % Numbering the nodes for creating a permutation
Npar=[];
for i=1:int16(2*N)
Np_rand=Np(randperm(length(Np)));
Npar=[Npar; 1 Np_rand 1];
end

Npar1=unique(Npar,'rows');

%% Calculate the fitness function
Ft=[];
for j=1:length(Npar(:,1))
R=0;
for i=1:N
t=Rd(XY(Npar(j,i),1),XY(Npar(j,i+1),1),XY(Npar(j,i),2),XY(Npar(j,i+1),2));
R=R+t;
end
Ft=[Ft; R];
end
Pft=Ft/sum(Ft);
[Pft,r]=sort(Pft,'ascend');  % Pft is the sorted in ascending order of fitness
```

```
    probability values and r is the index of the parent route
%k=randperm(length(a));

%%Selecting the parents based on fitness function values
t=int16(0.5*length(Pft));

Par=[];  % Matrix selected for parents
for i=1:t
Par=[Par;i Npar(r(i,1),:)];
end

%% Cross over algorithms being implemented
%PMX
CrossPMX=transpose(Par(:,1));
CrossPMX=CrossPMX(randperm(length(CrossPMX)));
for i=1:(length(CrossPMX)-1)
Bp=[(int16(0.2*(length(Npar)-1))) (int16(0.3*(length(Npar)-1)))];
end


K=[2 3 4 5 6 8];
B=K==3;
sum(B)
```

Dijkstra 算法解决最短路径问题

```
#include<iostream>
#include<cstdlib>
using namespace std;
const int inf=10000,NoN=6;
int matrix[NoN][NoN]={
  {0,7,9,inf,inf,14},
  {7,0,10,15,inf,inf},
  {9,10,0,11,inf,2},
  {inf,15,11,0,6,inf},
  {inf,inf,inf,6,0,9},
  {14,inf,2,inf,9,0},
  };
struct list
{
  int dist,dn,sn;
}output[20]={0};
int index=0;
struct linkedList
{
  int ns,nd,length;
```

```
  struct linkedList *link;
}*r=NULL,*f=NULL;

void Cities()
{
  string city[NoN]={"Bhubaneswar","Puri","Cuttack","Dhenkanal","Jajpur","
      Angul","Balangir","Kendujhar","Balasore","Rourkela"};
  cout<<"INDEX"<<"␣␣␣␣"<<"CITIES"<<endl;
  for(int i=0;i<NoN;i++)
  {
    cout<<"␣"<<i<<"␣␣␣␣␣"<<city[i]<<endl;
  }
}

int checkoplist(int node)
{
  for(int k=0;k<index;k++)
  {
    if(output[k].dn==node)
    return(0);
    else
    return(1);
  }
}

int checkll(int node,int edgelength)
{
  //find if given node is already in linkedlist.
  struct linkedList *p=r,*q=r;
  while(q!=NULL)   //didnt use t->link!=null because in that case last node
      wont be checked.
  {
    if(q->nd==node)
    {
      //if it exists, check whether new distance is less.
      //for first node
      if(q==r)
      {
        if(r->length>edgelength)
        {
          //If yes, delete pre-existing node.
          if(f==r)//suppose only one node exists and has only one adjacent.
              meh.lonely node.
          f=q->link;
          r=q->link;
          free(q);
          return(1);
```

```
        }
        else
        return(0);
      }
      //for last node
      else if(q->link==NULL)
      {
        if(q->length>edgelength)
        {
          while(p->link!=q)
          p=p->link;
          f=p;
          p->link=NULL;
          //If yes, delete pre-existing node.
          free(q);
          return(1);
        }
        else
        return(0);
      }
      //for intermediate node
      else
      {
        while(p->link!=q)
        p=p->link;
        if(q->length>edgelength)
        {
          //If yes, delete pre-existing node.
          p->link=q->link;
          free(q);
          return(1);
        }
        //else dont add node.
        else
        return(0);
      }
    }
    q=q->link;
  }
  return(1);
}

int shortestPath()
{
  //check shortest node to promote to output list.
  struct linkedList *a=r->link,*temp=r,*b=r;
  int min=r->length;
```

```
    //for lonely node.
    if(a==NULL)
    {
      output[index].sn=b->ns;
      output[index].dn=b->nd;
      output[index].dist=b->length;
      index++;
      r=NULL;f=NULL;
      return(inf);
    }
    else
    {
      while(a!=NULL)
      {
        if(a->length<min)
        {
          b=a;
          min=a->length;
        }
        a=a->link;
      }
    }
    output[index].sn=b->ns;
    output[index].dn=b->nd;
    output[index].dist=b->length;
    index++;
    //now to delete the node from linkedlist
    if(b==r)
    {
      if(f==r)
      f=b->link;
      r=b->link;
      free(b);
    }
    else if(b->link==NULL)
    {
      while(temp->link!=b)
      temp=temp->link;
      f=temp;
      f->link=NULL;
      free(b);
    }
    else
    {
      while(temp->link!=b)
      temp=temp->link;
      temp->link=b->link;
```

```
    free(b);
  }
  //symmetric matrix so changing certain values to zero.
  for(int j=0;j<NoN;j++)
  matrix[j][output[index-1].dn]=0;
  matrix[output[index-1].dn][output[index-1].sn]=0;
  return(output[index-1].dn);
}

void addAdjacent(int source)//this function accepts a node and adds its
    adjacents.
{
  int k,l=0;
  //adding adjacents of source to ll
  for(int i=0;i<NoN;i++)
  {
    if(source==inf)
    break;
    struct linkedList *t;
    t=(struct linkedList*)malloc(sizeof(struct linkedList));
    if(matrix[source][i]!=0 && matrix[source][i]!=inf)
    {
      //checking list if adjacent of source is already selected as shortest-
          path node, if not then continue into if block.
      if(checkoplist(i))
      {
        t->ns=source;
        t->nd=i;
        //distance from source node to current node
        for(k=0;k<index;k++)
        {
          if(output[k].dn==source)
          break;
        }
        //add source and destination to linkedlist along with added distance
        t->length=output[k].dist+matrix[source][i];
        t->link=NULL;
        //check if the added adjacent provides a shorter path. If yes, add,
            else dont.
        if(checkll(i,t->length))
        {
          if (f==NULL)
          {
            f=t;
            r=f;
          }
          else
```

```
            {
              f->link=t;
              f=t;
            }
          }
          else
          free(t);
        }
        else
        continue;
      }
    }
    if (source!=inf)
    {
      int nextNode=shortestPath();
      addAdjacent(nextNode);
    }
}

void display(int s,int d)
{
    int i=0;
    while(output[i].dn!=d)
    i++;
    cout<<"distance is "<<output[i].dist<<endl<<"Intermediate cities"<<endl;
    while(d!=s)
    {
      i=0;
      while(output[i].dn!=d)
      i++;
      string city[NoN]={"Bhubaneswar","Puri","Cuttack","Dhenkanal","Jajpur","
          Angul","Balangir","Kendujhar","Balasore","Rourkela"};
      cout<<city[output[i].sn]<<endl;
      d=output[i].sn;
    }
}

int main()
{
    int source,destination;
    Cities();
    cout<<"Enter index of start point and destination"<<endl;
    cin>>source>>destination;
    output[index].sn=source;
    output[index].dn=source;
    output[index].dist=0;
    index++;
```

```
    addAdjacent(source);
    /*for (int i=0;i<index;i++)
    {
      cout<<"op.dn"<<output[i].dn<<endl;
      cout<<"op.sn"<<output[i].sn<<endl;
      cout<<"op.dist"<<output[i].dist<<endl;
    }*/
    display(source,destination);
    return(0);
}
```