诚 信 声 明

我声明，所呈交的毕业论文是本人在老师指导下进行的研究工作及取得的研究成果。据我查证，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得其他教育机构的学位或证书而使用过的材料。我承诺，论文中的所有内容均真实、可信。

毕业论文作者签名：　　签名日期：　　年　月　日

一种量子遗传算法的设计实现与应用

[摘 要]

遗传算法模拟生物种群进化过程，被证明在解决许多 NP 问题时实用且灵活有效。然而受经典计算机的计算能力限制，遗传算法的随机搜索特性以及种群多样性需求会带来收敛速度慢或过早收敛等问题。而量子计算机利用量子计算机制能实现计算上的“并行”，从而可以加速遗传算法的进化过程，同时使更多个体参与进化，避免算法早熟收敛。本文基于量子算法领域最具代表性的算法之一——Grover 算法，尝试设计实现量子计算模型上的遗传算法形式，并使用 Q#语言实现算法，在经典计算机上仿真实验并对结果进行对比分析。

本文首先实现了 Grover 原始算法的一种变形——BBHT 最优查找算法，将之应用于 0-1 背包问题，并与经典遗传算法进行比较；其次为适应任意初始种群这一设置，研究设计了 Grover 算法具有任意初始态的推广形式；最后基于 Grover 推广算法，研究设计具有变异算子的量子遗传算法形式。通过实验数据分析以上各算法的表现，论证了量子计算机制能为经典遗传算法带来性能上的提高。

[关键词] 量子遗传算法；量子计算；Grover 算法；变异操作

Design of a New Quantum Genetic Algorithm and Its Application on 0-1 Knapsack Problem

Abstract：

Genetic Algorithm is an excellent solution to many NP problems in practice. However, when the scale of computation is getting large, it will have a poor performance in convergence property and will encounter a problem of premature. Quantum computer and Quantum computing may overcome such constraints that the genetic algorithm has to meet in the classical computing environments, by making use of both quantum mechanisms and quantum resources.

In this paper, we try to explore new forms of genetic algorithms in quantum computers, particularly considering the dramatic progress recently on technologies of quantum devices. We simulate our algorithms and do experiments in Q# programming language developers, trying to find evidences which can show the gap in computing power between quantum computing and classical computing.

First of all, we compare the performance of the BBHT algorithm, one variant of the well known Grover's algorithm, and a classical genetic algorithm, by applying them to the 0-1 knapsack problem. Next, we consider an arbitrary initial state, in order to make the Grover's algorithm be compatible to a setting of arbitrary initial population in traditional genetic algorithm researches. Based on this form of the Grover's algorithm, at last, we design a new quantum genetic algorithm with a mutation operator. Experimental results show that quantum mechanics can improve the performance of algorithms, particularly in searching efficiency and convergence property.

Keywords：Quantum Genetic Algorithms；Quantum Computation；Grover's Algorithm; Mutation Operation

目　　录

1 绪论

遗传算法在八十年代末作为实现最优算法和机器学习的一种方法得到人们的大量关注。遗传算法是受益于自然选择和生物进化过程而得出的一系列演算模型，能够模拟生物种群进化过程。它利用一个类似染色体的简单数据结构，把一个特定问题的潜在解决方案进行编码，然后利用重组算子对其进行操作以保护关键的信息[1]。遗传算法[2]基本上可以描述为用于寻找最优解的搜索算法，尽管其应用的范围十分之广阔。多项研究证明，遗传算法是解决大数据问题的灵活而有效算法[3]，并且被证明在解决许多 NP 问题时实用且灵活有效[4]；然而受经典计算机的计算能力限制，遗传算法的随机搜索特性以及种群多样性需求会带来收敛速度慢的问题；在收敛速度和全局搜索能力之间也存在矛盾，在算法的后期阶段，搜索效率很低，很容易收敛到局部最优解[5]。随着问题规模的扩大这些缺陷更加凸显。虽然已有很多研究进行了改进，但很难有本质突破。

而量子计算及量子计算机体系的思想在那十年成熟，使得在量子计算机上能够利用量子机制进行计算。量子计算机利用量子机制的重要特性，使用量子资源实现了计算上的“并行”，从而实现量子算法相对于经典算法的高加速比，十分显著地提升问题求解的速度。例如大数分解问题[6]，对于量子算法，该问题可以在多项式时间内解决；而在经典算法里这是个典型的 NP 问题；以及在非结构化数据库中的搜索问题[7]，量子算法能够对经典算法进行二次加速，即量子搜索算法时间复杂度为$O(\sqrt{N})$，而经典搜索算法时间复杂度为$O(N)$。本文计划首先实现 Grover 搜索算法的一个变形

——BBHT 最优查找算法，该算法能在$O(\sqrt{N})$时间内搜索出最大或最小值；之后将其应用于 0-1 背包问题。该问题是 NP 完全问题，传统遗传算法在该问题上能获得高效查找效率。本文将对比经典遗传算法和 BBHT 算法在同一问题上的查找效率。

遗传算法采用群体导向的搜索策略，这使得它易于大规模并行实现，可充分发挥量子计算机的计算能力。在遗传算法量子化的研究上，K.H.Han 等人提出在将遗传算法各要素量子化后在经典计算机上进行仿真，称为量子衍生遗传算法。该算法将量子态、量子门、量子状态特性、几率幅等量子概念引入到遗传算法当中。量子衍生遗传算法虽被证明在多个问题的优化求解上性能超出经典遗传算法[8～11]，但它存在经典计算环境下不可克服的缺陷，无法完全利用量子资源和量子机制特性，因此不能满足人们对高效量子遗传算法的要求。

真正发挥双方计算优势的量子遗传算法应该是在量子计算机上实现遗传算法各要素，找到量子计算模型上的遗传算法表现形式 [12～14]。当前，量子遗传算法的实现离不开 Grover 算法[6,13～14]；Grover 算法的最初形态是在解和非解构成的空间中进行全局搜索。借鉴经典遗传算法将搜索范围缩小到局部并采用遗传机制提高收敛效率这一思想，本文尝试在 Grover 搜索算法中引入遗传算子，从而缩小 Grover 算法搜索空间，进而考察搜索效率的改变。同时，由于量子态能实现任意个体的叠加，因此能扩大种群规模，使更多个体参与进化，从而丰富种群多样性，避免算法早熟收敛。

本文采用的是 2017 年末发布的基于 Microsoft Visual Studio 开发环

境的量子编程框架 Microsoft Quantum Development Kit；编程语言 Q#是一门领域专用语言，仅描述量子算法与过程，使得开发人员能在经典计算机和经典算法程序的控制下编写在量子处理器上执行的子程序。在量子处理器广泛运用之前，Q#的子程序都在模拟器上执行。在实际开发过程中，往往使用 C#语言负责驱动量子程序，Q#负责量子计算的模式；不仅能利用 C#丰富的函数库，同时在 Q#环境中集成了量子领域重要论文的算法，增强了代码的复用性；同时经典程序和量子算法的分离也更加符合量子计算机的系统结构。

本文尝试通过实现任意初态的 Grover 推广算法，并引入量子变异操作进行遗传演化，从而探索在非全局条件下 Grover 算法的搜索能力。有研究表明，Grover 算法里的初始态可能会因为门电路的酉误差而偏离均匀态，然而对于 Grover 推广算法，该误差并不会影响结果[15]。

距离量子计算机的面世并设计出能够在量子计算机上运行的遗传算法还需要时日，这也不禁让人思考量子遗传算法的可行性。本文针对此方面作出了一些探索，不仅将 Grover 算法应用于 NP 完全问题并和经典遗传算法进行效率比较，还实现任意初态的 Grover 推广算法和变异操作，对非全局条件下的 Grover 推广算法效率进行分析。随着量子计算机研发的高速进展，成果涌现如 D-Wave 量子模拟机、首颗量子科学试验卫星、高效量子存储器研发成功等。一旦量子计算机走入现实，量子遗传算法将获得更加广大的应用空间。

2 经典遗传算法

2.1 算法介绍

遗传算法是基于基因和自然选择的进化论的自适应性搜索算法。算法第一步即创造一个随机的初始种群，其中每个个体都用特定的编码表示，编码格式中包含着问题中的一些描述和特征，把一个特定问题的潜在解决方案进行编码。第二步用适应值函数来评估种群中的每个个体，依据适应值的不同来进行种群的迭代操作。第三步从种群每一代中选出两个高适应值的个体进行交叉和变异操作，作为父代产生两个子个体，并将生成的子个体随机替换种群中的两个个体。种群不断迭代直到种群的总适应值达到了一个特定的阈值或者迭代次数达到了一个预设的值。算法的基本执行过程如下[16]：

（1）初始化：确认种群规模数N、交叉概率P_c、变异概率P_m和种群演化终止条件

（2）个体评估：利用适应值函数计算种群$T(i)$中每个个体的适应值。

（3）种群演化

A） 选择：从种群$T(i)$中利用选择操作选择出$M/2$对父代个体$(M \geq N)$

B） 交叉：选择出$M/2$对父代个体，按照一定的交叉策略和交叉概率P_c执行染色体交叉操作，形成M个中间个体。

C） 变异：对产生的M个中间个体独立地依据预设变异概率P_m进行变异，形成M个候选个体。

D） 选择（子代）：利用适应值函数从上述M个候选个体中选择出N个个体形成新一代的种群$T(i+1)$。

（4）终止检验：如果此时种群$T(i+1)$满足了种群演化终止条件，则种群$T(i+1)$中具有最大适应值的个体就作为算法的最优解，并终止迭代；否则令$i=i+1$，转步骤（3）

遗传算法以一定基数的染色体（通常是随机的）开始实现。而后对其结构进行评估并分配繁育的机会。代表着问题更优解的染色体比相对较差解的有更大的机会进行繁育，解的好坏通常根据当前种群进行定义。

在遗传算法中，问题的适应值函数使得种群向着解的方向演化。在复杂的问题中，通常不太可能得到一个能准确地反映问题本质的适应值函数，故使用不精确的适应值函数是不可避免的。这表明在选择的过程中不能够因为相差无几的适应值而判断基因型的好坏。更好的方法是选取高适应值个体的一部分并将其用于新个体的生成。这种选择算法叫做截断选择法。在种群迭代过程中，每一代都会产生一个新的种群从而代替旧有的种群。在种群递增过程（稳定态）中，每次仅生成两个子代并加入到种群中，这也是解决多适应值函数问题时我们需要的方法。

2.2 算法应用

遗传算法作为一种具有较好全局收敛性的随机优化搜索算法，在生物计算、模式识别、图像处理、工程计算等众多领域取得了很好的应用；同时在经典问题如求解布尔方程组问题中，传统最佳线性逼近的算法是基于穷举的方法，解空间规模呈指数增长，而遗传算法对方程组问题的求解是

确实有效的一种算法，利用改进策略后对特定问题有很好的求解成功率和很快的收敛速度[17]。

而本文所要讨论的 0-1 背包问题是一个具有代表性的组合优化问题，也是计算机科学中一个被证明的典型 NPC 问题[18]。背包问题的重要之处在于，许多整数规划问题的算法执行过程中都需要使用一个高效的求解背包问题的算法来辅助运算；同时背包问题在许多领域方面也能得以实际应用。背包问题的求解主要有两种：一种是精确性算法，可以得到问题的精确解，但是算法的时间复杂度呈现出指数级的增长态势，因此只能用于求解小规模问题；另一种是启发式算法，虽不一定能够得到问题的最优解，但是可以在较短时间里得到问题的有效满意解，因此随着计算机技术和数据挖掘技术的发展，启发式算法逐渐成为解决较大规模 0-1 背包问题的一种主流方法，而其中遗传算法是一种比较优秀的启发式优化算法，在解决大规模组合优化问题上表现出较强的优势[19]。

然而，遗传算法是以一个不断迭代来寻找最优值的过程，但该算法对新空间的搜索能力是有限的，处理规模也很小，特别是在优化的后期阶段，搜索效率很低，很容易收敛到局部最优解；同时选择算子，交叉算子和变异算子的实现也需要很多参数，如交叉率和变异率，且这些参数的选择严重影响了解的质量；遗传算法对初始种群有很强的依赖性，直接影响着解的收敛性和优化结果的质量；根据一般情况，遗传算法迭代次数越多，算法的收敛性越好；然而在增加遗传迭代次数的同时，算法的计算工作量增加，这消耗了大量的计算时间[20]。在经典计算机上实现的遗传算法虽然

仍在发展，但是不可避免地遇到了早熟收敛和算法效率上的问题。

3 量子计算

3.1 量子计算基础

3.1.1 Hilbert 空间

公理一　任一孤立物理系统都有一个称为系统状态空间的复内积向量空间（即 Hilbert 空间）与之相联系，系统完全由状态向量所描述，这个向量是系统状态空间的一个单位向量。

量子计算中信息的基本单元是量子比特。每个量子比特都是一个二级量子系统，在二维 Hilbert 空间表示为一个单位矢量；设$|0>$和$|1>$构成这个状态空间的一个标准正交基，则状态空间中的任意状态向量可写作

$$|\psi>=a|0>+b|1>, \qquad |a|^2+|b|^2=1$$

在 Hilbert 空间里我们把两个基本态标示为$|0>$和$|1>$；并把二级量子系统描述为基本态的叠加态，每个二级量子系统最终只能是基态$|0>$或$|1>$中的一个。其中，$\alpha,\beta\in\mathbb{C}$叫做量子相位，分别代表着测量到叠加态结果$|0>$或$|1>$的概率的平方根，且$\|\alpha\|^2+\|\beta\|^2=1$。线性组合结构的量子比特称为量子寄存器，编码着所有可能经典状态的叠加态。对于有n个量子比特的量子寄存器，其对应的状态是2^n-维 Hilbert 空间中的标准态：

$$|\psi>_r=\sum_{i=0}^{2^n-1}a_i|i>,\ \text{其中}\Sigma_{i=0}^{2^n-1}|a_i|^2=1,\ i\in\mathbb{N}$$

3.1.2 演化

公理二　一个封闭量子系统的演化可以由一个酉变换来刻画。即系统在时刻t_1的状态$|\psi>$和系统在时刻t_2的状态$|\psi'>$，可以通过一个仅依赖于时间t_1和t_2的酉算子U相联系：

$$|\psi' >= U|\psi >$$

公理二描述了封闭量子系统的量子状态在两个不同时刻的关系。量子系统的演化是由应用于量子比特的特定线性算子、酉算子U构成的。运用量子线性算子的重要结果就是基态$|0>$和$|1>$计算后的线性组合，这也叫做“量子并行性”。而在经典的二级系统里，我们必须要分别计算两个可能状态$|0>$和$|1>$。

有几个在量子计算与量子信息中非常重要的单量子比特上的酉算子，例如 Pauli 矩阵、X 矩阵（也叫量子非门）、Hadamard 门等。同时任意的多量子比特门都可以由受控非门（controlled-NOT 或 CNOT 门）和单量子比特门复合而成，所以可以说受控非门和单量子比特门组成了一个封闭的量子系统演化模型。

3.1.3 量子测量

公理三　量子测量由一组测量算子$\{M_m\}$描述，这些算子作用在被测系统状态空间上，指标m表示实验中可能的测量结果。

如果想要从量子系统传递信息到经典系统里，我们就必须对量子状态进行测量，而测量的结果是基于概率的：我们有$|\alpha|^2$的几率得到状态$U|0>$和$|\beta|^2$的几率得到状态$U|1>$。量子的不可复制性[21]表明我们不可能从量子状态$|\psi>$的若干份拷贝中的系数α和β中得到完整的信息。

3.1.4 复合系统

公理四　复合物理系统的状态空间是分物理系统状态空间的张量积，

若将分系统编号为 1 到n，系统i的状态被置为$|\psi_i>$，则整个系统的总状态为$\psi_1 \otimes \ldots \otimes \psi_n$。

设V和W是维数分别为m和n的向量空间，并假定V和W是 Hilbert 空间，于是$V \otimes W$是一个mn维向量空间，$V \otimes W$的元素是V的元素$|v>$和W的元素$|w>$的张量积$|v> \otimes |w>$的线性组合。所以对于复合系统，相应地有某个状态可以记作$|v> |w>$，属于联合系统VW。应用叠加原理到这种乘积形式的状态，就得到如上提出的张量积。

叠加原理使得量子寄存器得以在相同空间的前提下比经典寄存器存储更加多的信息：N 位的经典比特能够存储2^N个信息中的其中一个，对应的量子比特则处于2^N个信息的叠加态中。经典寄存器上的操作能产生一个结果，而量子寄存器上的操作能够产生所有可能结果的叠加态。这就是前文提到的“量子并行性”实际含义。

同时，量子寄存器在观测后坍塌到其中一个状态，叠加态不复存在。看似没有什么用，其实这取决于采取的算法；问题的叠加态可能有共同的特征。如果能够通过测量确定这些特征并循环执行算法，有概率能够得到问题的答案。这也是知名量子算法的工作原理。首先，我们要生成量子叠加态并应用相应的操作，然后对叠加态执行傅里叶变换来推测所有的共同特征；最后重复操作直到变换得到的信息由足够的可信度。

量子的线性机制中另外一个重要的特征就是纠缠态。经典系统中AB组合的状态是由其子系统决定的，而量子系统中组合系统的状态是由子系统状态的张量积$\otimes$决定的，所以组合系统$|\psi>_{AB}$的状态可以表述为

$$Bell_{AB}=\frac{1}{\sqrt{2}}[|O>_A\otimes|O>_B+|1>_A\otimes|1>_B]$$

纠缠态是两个处于叠加状态的量子比特的一种量子联系。例如前文中提过的例子，我们可以通过计算得到一个处于$|0>$和$|1>$叠加状态的量子比特，对该量子比特的会使其从叠加态坍塌到一个经典状态。纠缠态使得原有的叠加态比特和最终结果的叠加态能有量子意义上的关联，故当我们对最终结果进行测量使其坍塌到其中一个解，原有的叠加态比特也会坍塌到经典状态（值 0 或 1）。并且结果的叠加态会坍塌到所有可能的经典值之一上，产生一个测量后的结果。

3.2 Grover 搜索算法

3.2.1 Grover 算法介绍

算法思想

该算法解决的是在非结构化数据库中的搜索问题。Grover 算法的时间复杂度是$O(\sqrt{N/t})$，其中N是数据库中的总空间大小，t是可能解的数目。然而，经典算法要解决搜索问题时必须要遍历数据库中的所有个体直到找到一个解，其时间复杂度为$O(N/t)$。Grover 算法的基本思想是放大与解相关联项的相位，同时缩小那些与解无关项的相位。算法执行过程中进行了$O(\sqrt{N/t})$次酉操作，而后对量子比特进行测量能够以较高的概率得到一个可能解。Grover 算法对经典算法的加速是基于非结构化数据库的，否则，利用二叉树查找经典算法可以达到$O(\log N)$的时间复杂度。值得注意的是，经典算法中总是遍历所有个体以得到数据库中的所有解，而量子测量中的

不确定性使得量子算法能够在数据库的若干解中随机选择其一。然而如果重复执行整个量子搜索算法，有可能会得到其他不同的解。

算法描述

1）Oracle

我们在 Grover 算法中引入基于黑箱（oracle）的搜索算法，这样就能给出搜索过程的非常一般化的描述，并能以集合方式形象地表示它的步骤和执行方式。设有一个量子 oracle 可以识别搜索问题的解，识别的结果通过 oracle 的一个量子比特给出，更准确地来说该 oracle 是一个酉算子，根据在计算机基的作用定义

$$|x>|q>\xrightarrow{o}|x>|q\oplus f(x)>$$

其中$|x>$是一个指标寄存器，$\oplus$代表模 2 加法，oracle 的量子比特$|q>$是一个单量子比特，当$f(x)=1$时反转，否则不变。我们可以通过制备$|x>|0>$，应用 oracle，检查 oracle 量子比特是否翻转到$|1>$，来判断x是否为搜索问题的一个解。

在量子搜索算法中，在 oracle 量子比特初始化为$(|0>-|1>)/\sqrt{2}$时，应用 oracle 时有用的。若x不是搜索算法的解，应用 oracle 到状态$|x>(|0>-|1>)/\sqrt{2}$，并不改变状态；另一方面，如果x时搜索算法的解，则$|0>$和$|1>$在 oracle 的作用下相交换，给出终了状态$-|x>(|0>-|1>)/\sqrt{2}$。于是 oracle 的作用是

$$|x>\left(\frac{|0>-|1>}{\sqrt{2}}\right)^{o}\rightarrow(-1)^{f(x)}|x>\left(\frac{|0>-|1>}{\sqrt{2}}\right)$$

注意 oracle 量子比特的状态没有改变，事实上这个量子比特将在量子搜索算法过程中保持为$(|0>-|1>)/\sqrt{2}$。

在这个约定下，oracle 的作用可以写成

$$|x>\xrightarrow{O}(-1)^{f(x)}|x>$$

我们说 oracle 通过改变解的相位，标记了搜索问题的解。对有M个解的N元搜索问题，实际上为得到一个解，只需要在量子计算机上应用搜索 oracle $O\left(\sqrt{N/M}\right)$次。

2）Grover 迭代

量子搜索算法由反复应用记作G的称为 Grover 迭代（Grover iteration）或 Grover 算子的量子子程序组成。Grover 迭代的量子线路如下图所示，可分为四步：

Grover迭代过程
1. 应用oracle O 2. 应用Hadamard变换$H^{\otimes n}$ 3. 在计算机上执行条件相移，使|0>以外的每个计算基态获得-1的相位移动 $$\|x>\rightarrow-(-1)^{\delta_{x0}}\|x>$$ 4. 应用Hadamard变换$H^{\otimes n}$

（算法 3.1）

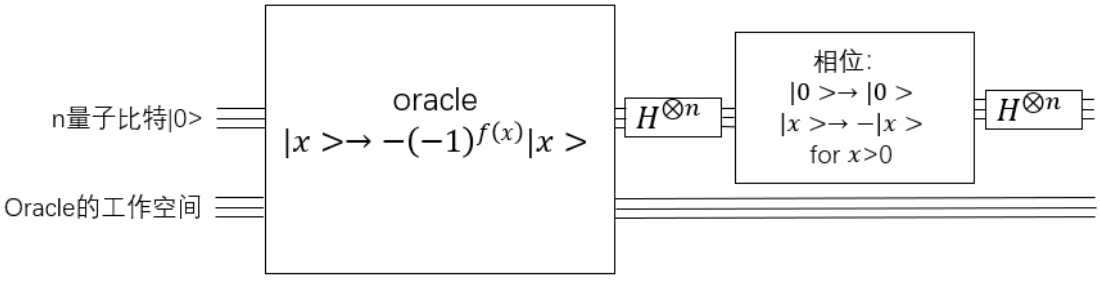


图 3-1 Grover 迭代 G 的线路

Grover 迭代的作用是变换量子基态的概率幅，使得计算基中的一个观测能以很高的概率产生于搜索问题解相关的量子态。

3）算法框架

搜索算法运行的框架如下图所示，算法使用一个单个的n量子比特寄存器。oracle 的内部工作，包括它可能需要的附加工作量子比特，对描述量子搜索算法本身并不重要。算法的目的是用最少的 oracle 应用次数求出搜索问题的一个解。

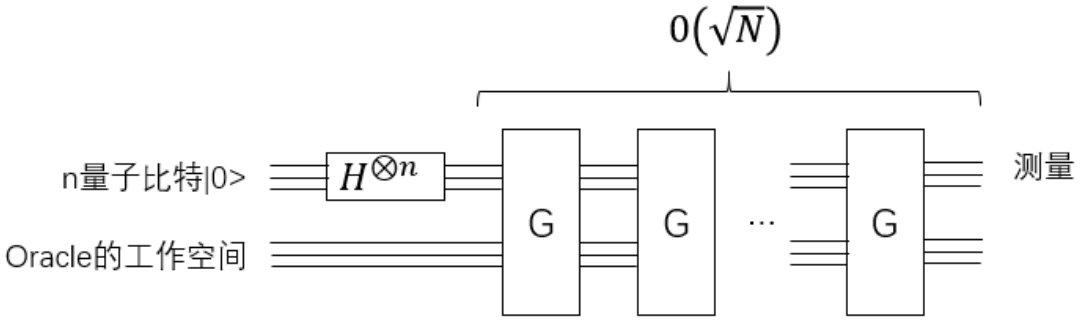


图 3-2 量子搜索算法的线路框架

算法从计算机的初态$|0>^{\otimes n}$开始，用 Hadamard 变换使计算机处于均匀叠加态

$$|\psi>=\frac{1}{N^{\frac{1}{2}}}\sum_{x=0}^{N-1}|x>$$

Grover 算法重要之处在于其使得基于 oracle 的量子算法能彰显出相对经典算法的高效性，可以加速许多（虽不是全部）启发式搜索的经典算法[15]。

3.2.2 Grover 推广算法

1）解的个数未知的 Grover 算法

当解的个数t不定时，oracle 就无法确定需要标记的解个数。Michel Boyer 的研究得出的 BBHT 最优查找算法（算法 4.1）能够解决解个数未知的情况；算法取$\sqrt{N}$作为迭代上界，并在算法起始时随机选取非负整数$M < \sqrt{N}$，如果M次迭代过程中找到了解，算法停止；若M次迭代仍然没有找到解，那么就把M作为迭代的下界，并且随机选取非负整数M'且满足$M < M' < \sqrt{N}$；BBHT 算法在解的个数t不定找到一个解的时间是$O(\sqrt{N/t})$。

2）初态为非均匀态的 Grover 算法

在 Grover 算法的基础上，我们可以对其进行推广：当初态并非均匀态，而是任意分布时：

$$|\psi_{ran}> = \sum_{x=0}^{N-1} P_x |x>$$

其中P_x即为相应量子状态的几率幅。针对随机初始态$|\psi_{ran}>$进行 Grover 迭代，根据不同的初始态，算法的最大查找成功率和所需要的迭代次数都会相应地变化。值得注意的是为了得到最优迭代数，必须要知道 oracle 标记解数和初始态里解与非解的几率幅；同时另外一个重要的结论是 Grover 推广算法仍然需要以时间复杂度$O(\sqrt{N/t})$搜寻到一个解[15]。

3.3 量子编程语言

在 Q#面世前，已有若干量子计算编程语言可供开发人员选择，如 QCL、Liquid 等。

1）量子程序设计语言（Quantum Computing Language，QCL）

QCL 是遵循 GPL 条款的开源编程语言。其程序结构上是结构化命令式量子程序设计语言,在传统面向过程编程语言的基础上添加了负责量子操作的函数和量子过程。而 QCL 编译器采用的是 qlib 仿真库，所以量子算法执行过程中所有 qubit 的量子态都可以被观测到，这在真正的量子计算机上是不可能实现的，故仅能作为一种量子算法的模拟。

2）Liquid

Liquid 是由微软提供的开源量子模拟器，主要包含：基于 F#的编程语言、编译器、模拟器、周边服务及后端；它提供了大量的高级操作来简化量子编程，如受控门与反计算的自动实现，在其发布的时期算是一个可快速上手的可靠模拟器。而 Liquid 因其公开发布版模拟 qubit 数不能超过 22 个，且项目仅停留在软件层面，使得开发人员想要真正的控制量子比特只能选择其他编程语言。

3）量子编程语言 Q#

2017 年末，微软推出量子编程语言 Q#，明确定义了完整的编程模型；过去的编程语言主要关心的是在 qubit 上做什么事以及怎么做，但是最终的量子计算机中不可能只有 qubit，也会有传统的 CPU 与 GPU，那么也就会遇到相应的问题：不同的计算部件之间如何进行交流？在计算过程中部件之间的调用关系？CPU 该如何调用量子芯片进行特定的计算？虽然经典操作与量子操作在编程语言中可以写在一起，但是如何在硬件上，尤其是如何在控制体系结构上实现这种混合计算是没有规范定义的；其中不规范性

包括未定义能够调度的资源、未定义不同部件之间的交流方式等等。

然而 Q#是一门领域专用语言，仅用于描述在量子芯片上可能发生的操作，而对于在传统 CPU 上的操作，Q#的编程模型规定需要额外的比经典编程语言进行描述，如 C#或 Python。在本文实现的算法中，主程序基于 C#编程语言编写，而后将量子核心算法交给量子芯片（kernel，量子内核）执行；这个核心算法基于 Q#编程语言。所以，C#描述的部分最终会在传统 CPU 上执行，而 Q#描述的内核最终由量子芯片以及与之配套的控制芯片来执行。简而言之，量子计算机在 Q#的定位是一个量子加速器，只是异构计算体系结构中的一部分。这种 C#语言负责驱动量子程序，Q#语言负责量子计算的模式使得开发人员能够专注设计算法和解决问题，降低开发难度，方便开发人员的人工和对程序进行分析。

4 0-1 背包问题的 Grover 算法实现

4.1 0-1 背包问题

问题描述：给定n个重量为$w_1, w_2, \cdots, w_n$，价值为$v_1, v_2, \cdots, v_n$的物品和容量为W的背包，求这个背包收纳物品的一个最优物品子集（每个物品仅选取一次），使得在不超过背包最大容量的前提下使得包内物品的总价值最大。

背包问题是一个 NPC 的问题。相对于传统的算法（如动态规划算法、回溯算法等），使用遗传算法求解背包问题能最快找出较优的解[22]。

由于每件物品只有装入与不装入背包两个选项，故采用二元表示法：$x_i = 0$是代表i个物品不装入背包，$x_i = 1$是代表i个物品装入背包，故 0-1 背包问题的数学模型可以表示为

$$max\ f\ (x_1, x_2, \dots, x_n),$$

4.2 0-1 背包问题的经典遗传算法实现

在第二节中已经介绍过一般经典遗传算法的迭代过程，这里简单介绍应用于 0-1 背包问题的算法策略。

（1）基因编码：基因由n个物品状态组成，1 表示装入，0 表示不装入，同时每个个体还有适应值、选择概率、累积选择概率。

（2）适应值函数：计算当前种群中所有对象的适应度，$f_i = \sum_{i=1}^{N} x_i \cdot v_i$

（3）选择算子：简单轮盘赌方式，首先计算种群中所有个体的选择概率和累积概率，然后利用随机数进行轮盘赌，挑出幸运者作为新种群。

（4）交叉算子：采用多点交叉策略，对两个随机选中的个体基因进行交换，基因交换的位置和个数都是随机的，使得个体的基因更具有随机性。

（5）变异算子：采用均匀变异的策略，选中个体基因变异的个数与位置都是随机选择的，并以预设的变异概率p_m进行变异操作。

（6）演化终止条件：当繁衍的过程中有t代都出现了相同的结果时，或是达到了预设迭代次数时算法终止。

经典遗传算法解决背包问题有各种改进算法，这些改进算法在不同的场景或前提下有着更加优秀的性能，如将选择概率固定化的顺序选择遗传算法（SBOGA）、使种群脱离“早熟”的大变异遗传算法（GMGA）、减小交叉交换基因量的双切点交叉算法（DblGEMA）等等；本文中用到遗传算子为比例选择模式、多点交叉与均匀变异，设置了一个对超出阈值的基因进行“惩罚”的适应值函数，并且利用简单轮盘赌方式进行选择，实现代码见附录 A。

4.3 0-1 背包问题的 BBHT 量子算法实现

4.3.1 BBHT 算法介绍

预先知道解的个数时，我们可以用 Grover 算法来找到其中一个解。然而如果不知道 Oracle 算法中标记的解的个数时，Grover 原始算法就行不通了。行不通的原因是在放大相位的过程中我们无法计算出 Grover 迭代的次数，从而就不能使与解相关项的系数变到最大。然而，即便解的数目 t 是事先不知道的，只要有可以标记解的 oracle 算法，还是可以使用 BBHT 算法在一组数据项$\{Ti\}_{i=0,1,\cdots,N-1}$中找到一个解。BBHT 算法能够以时间复杂度

$O(\sqrt{N/t})$找到一个解[23]。

BBHT算法过程
1. 初始化oracle，且$O\|i>=(-1)^{F(i)}\|i>$ 2. 初始化基准值$m=1$，$\lambda=6/5$，创造初始态$\|\psi_0>=H^{\otimes n}\|0>=\frac{1}{\sqrt{n}}\Sigma_j j$ 3. 均匀随机地选取小于基准值m的i个个体 4. 从初始态$\|\psi_0>$对种群i开始进行Grover算法迭代 4. 对量子比特进行测量，得到结果o 5. 如果测量结果To是一个解，则结束 6. 否则，$m=\min(\lambda m,\sqrt{N})$，返回步骤2

（算法 4.1）

4.3.2 BBHT 应用于 0-1 背包问题

由 BBHT 算法我们可以将其应用于经典 01 背包问题，并和经典遗传算法进行比较分析。

BBHT算法解决0-1背包问题
1. 生成随机初始个体m，创造初始均匀态$\|\psi_0>=H^{\otimes n}\|0>=\frac{1}{\sqrt{n}}\Sigma_j j$ 2. 选取适应值大于m的i个个体进行标记 3. 从初始态$\|\psi_0>$开始，对种群i进行Grover算法迭代 4. 对量子比特进行测量，得到结果o 5. 如果测量结果o优于当前个体则将其取代，并淘汰标记个体中适应值低值个体 6. 迭代次数是否已经达到$O\left(\sqrt{N/t}\right)$，是则算法结束 7. 标记个体中是否只剩下一个最优解，是则算法结束，返回步骤2

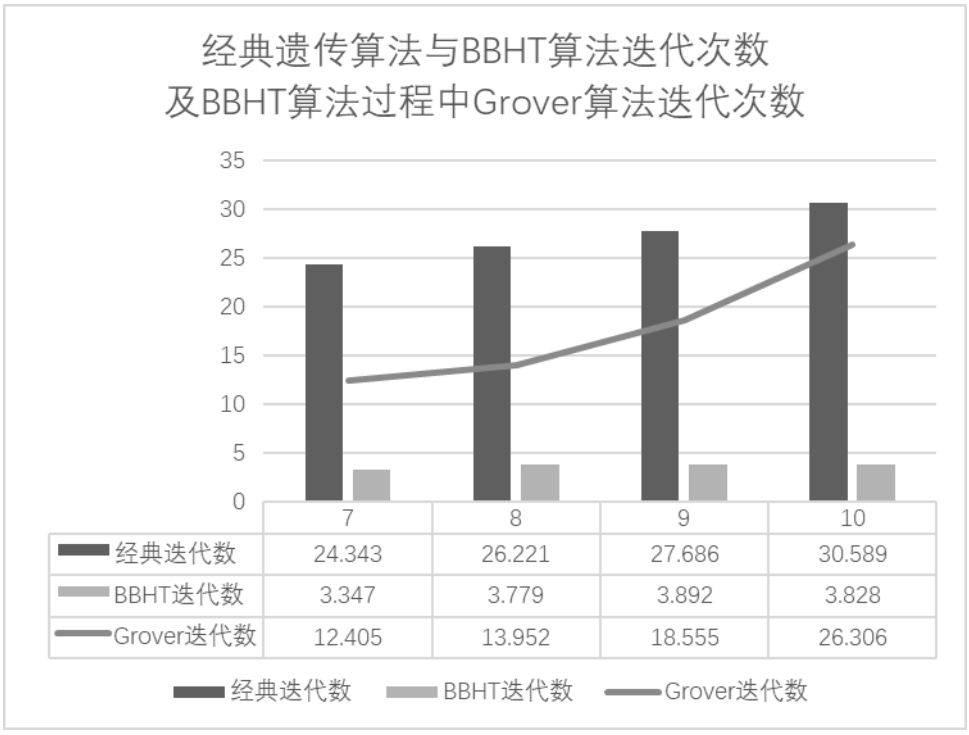
（算法 4.2 见附录 B）

4.3.3 两种算法实现的比较

量子计算的优异之处在多量子比特计算时才得以显现，量子状态矢量随着量子比特的数量呈指数级增长。模拟五十个量子比特已经接近当今的超级计算机的极限，每增加一个额外的量子位来增加计算的规模，就使得

存储所需要的存储空间加倍，并使计算时间大概翻一倍。但由于量子计算机并未面世，而受 Microsoft Visual Studio 中提供的 Q#语言效率限制，我们在问题中采用低于 10 位量子比特进行实验。

实验分别采用 7、8、9、10 位比特对 01 背包问题进行求解，并在重复一千次实验后计算出最终结果的平均迭代数、最终个体的平均适应值和问题最优解的命中率，如下表结果所示：



	7	8	9	10
经典迭代数	24.343	26.221	27.686	30.589
BBHT迭代数	3.347	3.779	3.892	3.828
Grover迭代数	12.405	13.952	18.555	26.306

表 4-1 经典遗传算法与 BBHT 算法迭代次数对比分析及 BBHT 算法过程中 Grover 算法迭代次数

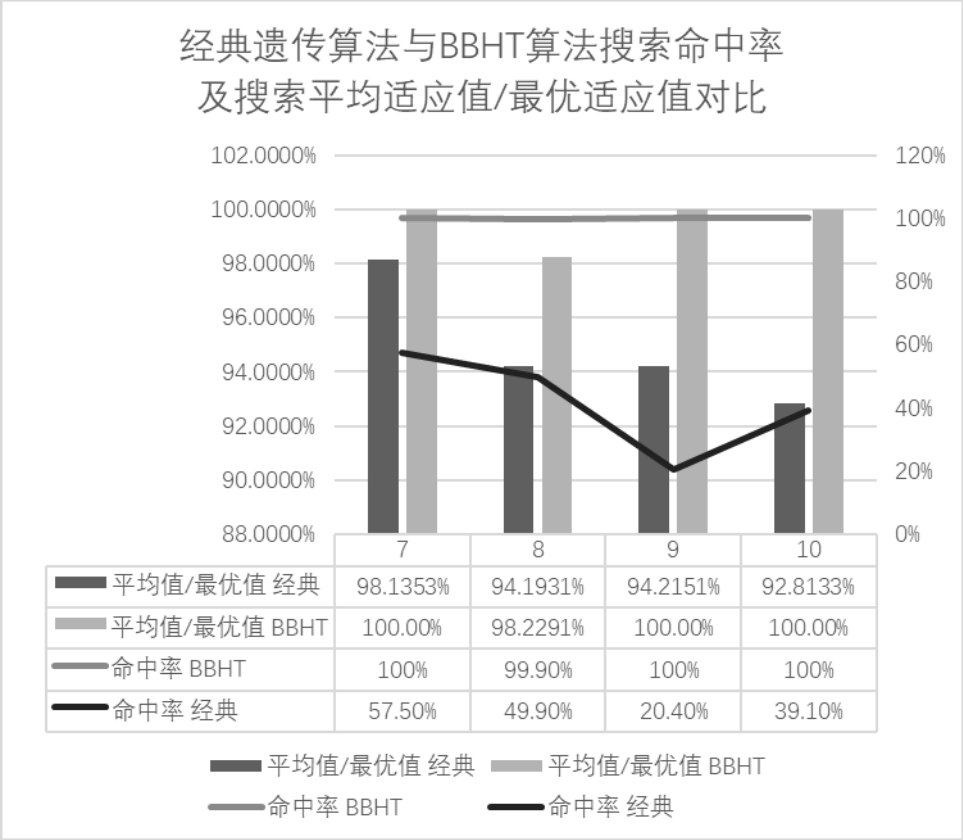


表 4-2 经典遗传算法与 BBHT 算法搜索命中率对比

及搜索平均适应值/最优适应值对比

由上两表可知 BBHT 相对于经典遗传算法在算法收敛速度方面有着显著的优势，同时算法的命中解的成功率也远高于经典遗传算法。值得注意的是，BBHT 算法中的 Grover 搜索是全局搜索，Grover 迭代是在所有个体的均匀叠加态上进行的，如果有量子计算机实体时，算法的运行将有很高的效率；然而在量子计算机未面世前，如果要用经典计算机模拟 Grover 迭代进程，仍需耗费大量计算机资源；并且随着数据库规模的扩大，算法效率将难以满足问题需要，变得不可行。

但同时通过数据也可以发现，受限于全局搜索的 Grover 算法，BBHT 算法运行过程中 Grover 的迭代次数是算法效率的一个不可忽视的因素，而与

此同时经典遗传算法虽然得最优解成功率不如人意，但是最终个体得平均适应值能够达到局部最优，与 BBHT 算法的平均适应值间没有显著的差距。由此启发我们的思考，对 Grover 是否可以通过将初始态从均匀态修改为由任意个数个体叠加而成的初始态从而进行搜索；如果在 Grover 迭代过程中搜索表现不好，则引入变异算子，对叠加态中的个体进行变异操作从而提高算法的搜索效率及加快收敛速度。

5 量子计算模型上的遗传算法

5.1 初始态为任意叠加态的 Grover 算法

5.1.1 算法性质

在经典 Grover 算法中，经 Hadamard 变换后初始态变为$|\psi_0>=H^{\otimes n}|0>=\frac{1}{\sqrt{n}}\Sigma_j j$，而 Grover 推广算法对初始态采用一定的随机策略，利用量子门进行量子态的修改，使得每个个体的几率幅随机化，达到初始态随机的目的。在修改初始态后，算法每次进行搜索解需要的迭代次数及其相应命中率是不可忽视的因素，并且迭代次数和命中率都会随着每次初态的改变而改变，下面我们讨论在随机初始态$|\psi_{ran}>$的情况下，需要多少次迭代以最大概率得到一个问题的解。

设$k(t)$为 t 次 Grover 迭代之后解的几率幅，$l(t)$为 t 次 Grover 迭代之后非解的几率幅。从 M.Boyer 等人的研究[24]中可知 t 次迭代之后对应问题解的量子态几率幅增长到$k(t)=sin\left[\omega\left(t+\frac{1}{2}\right)\right]/\sqrt{r}$，其中$\omega=2\,arcsin\left(\sqrt{\frac{r}{N}}\right)$，r 为解的个数，N 为问题空间大小。同时对应非解的量子态几率幅下降到$l(t)=cos\left[w\left(t+\frac{1}{2}\right)\right]/\sqrt{N-r}$[15]。对于$N\gg r$情况下的问题来说，最佳的测量时机是在$T=O\left(\sqrt{N/r}\right)$次迭代之后，此时$k(t)$达到最大值。

而在修改初始态的 Grover 算法中，算法迭代流程仍和前文中描述相同，初始态可以表示为$\Sigma k_i(0)+\Sigma l_i(0)$，代表着解和非解的几率幅组合。在 t 次

迭代后测得解的概率为

$$P(t) = P_{av} - \Delta P \cos 2[wt + Re(\phi)],$$

$$\text{其中} P_{av} = 1 - (N-r)\sigma_l^2 - \frac{1}{2}\left[(N-r)\left|\overline{(l(0))}\right|^2 + r\left|\overline{(k(0))}\right|^2\right],$$

$$\Delta P = \frac{1}{2}\left|(N-r)\overline{(l(0))}^2 + r\overline{(k(0))}^2\right|,$$

$$\sigma_l^2 = \frac{1}{N-r}\sum_{i=r+1}^{N}\left|l_i(0) - \overline{l(0)}\right|^2,$$

σ_l^2代表初始态下非解几率幅的方差，

故当$\cos 2[wt + Re(\phi)] = -1$即当$wt + Re(\phi) = \frac{\pi}{2}$时，概率$P$有最大值

$P_{max} = P_{av} + \Delta P$

然而ϕ的值大小决定了能够得到最大概率值P_{max}的迭代次数值t，下面我们计算ϕ的大小：

$$\overline{k(0)} = \frac{1}{r}\sum_{i=1}^{r} k_i(0), \qquad (1)$$

代表初始态下r个解的几率幅平均值，

$$\overline{l(0)} = \frac{1}{N-r}\sum_{i=r+1}^{N} l_i(0), \qquad (2)$$

代表初始态下$N-r$个非解的几率幅平均值。

并且根据欧拉公式可得

$$e^{2i\emptyset} = \cos 2\phi + i\sin 2\phi = \frac{f_+(0)}{f_-(0)} = \frac{\overline{l(0)} + \mathrm{i}\sqrt{\frac{r}{N-r}}\overline{k(0)}}{\overline{l(0)} - \mathrm{i}\sqrt{\frac{r}{N-r}}\overline{k(0)}}, \qquad (3)$$

由此联立（1）（2）（3）可以得出ϕ值大小，那么要取得概率最大值P_{max}时有

$$t = \frac{\frac{\pi}{2} - Re(\phi)}{w}$$

$Re(\phi)$代表的是ϕ的实数部分，而实际上当$\overline{l(0)}/\overline{k(0)}$是实数的时候，$\phi$也应该是实数。

由此我们可以得知在随机初始态$|\psi_{ran}>$的情况下，我们能够用t'次迭代以概率P'搜索到一个解。

5.1.2 算法实现

前文已经提及，受控非门和单量子比特 Hadamard 门能够构成一个封闭的量子系统，也就是说利用这两个门就可以生成由任意个体叠加而形成的初态，也就是我们所需要的随机初始态。本文在生成随机初始态方面的算法策略如下：

Grover推广算法生成随机初始态过程
1. 随机生成长度不超过n的序列对$cnot_i = \langle left, right \rangle$，其中的随机参数$left$与$right$代表进行变换的量子比特下标 2. 随机生成长度不超过n的数组$h_i = k$，其中随机参数k代表进行变换的量子比特下标 3. 在Grover迭代的$t(0)$时刻，遍历$cnot$数组，执行$CNOT(left_i, right_i)$，使量子比特随机纠缠 4. 遍历h数组，执行$H(h_i)$，进行随机均匀化

（算法 5.1）

由以上算法，我们就得到了随机初始态，并将其应用于 Grover 算法，则随机初始态的 Grover 推广算法如下：

随机初始态的Grover推广算法迭代过程

1. 根据算法5.1，得到随机初始态$|\psi_{ran}>$
2. 通过$O\left(\sqrt{N/t}\right)$次迭代完成搜索过程
   2.1 应用oracle算子检验每个元素是否为搜索问题的解
   2.2 应用Hadamard变换$H^{\otimes n}$
   2.3 在计算机上执行条件相移，使|0>以外的每个计算基态获得-1的相位移动

$$|x>\rightarrow -(-1)^{\delta_{x0}}|x>$$

   2.4 应用Hadamard变换$H^{\otimes n}$

（算法 5.2 见附录 C）

5.1.3 算法表现

实验分别采用 6、7、8 位量子比特对随机初始态 Grover 推广算法的命中率与对比 Grover 算法的加速比（即原始 Grover 算法收敛速度与 Grover 推广算法收敛速度的比值）进行统计分析，制表如下：

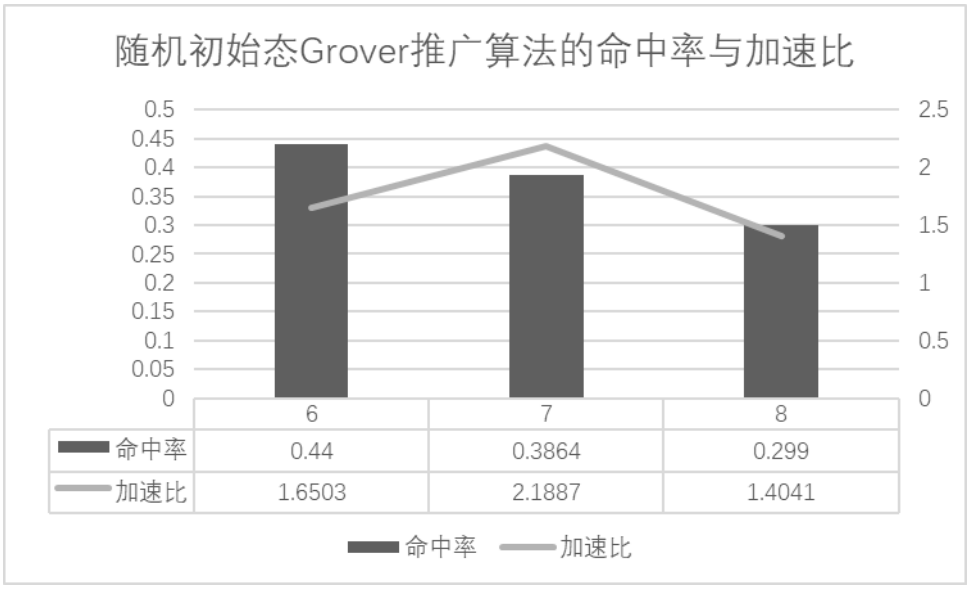


表 5-1 随机初始态 Grover 推广算法的命中率与加速比

从表中不难看出，Grover 推广算法相比 Grover 算法有着一定的加速比，但是受限于随机初始态的生成策略，当量子比特逐位增加时，量子系统中

的量子状态成倍增加，导致难以尽可能地随机挑选量子比特使其处于纠缠态中，同时也导致了当量子比特增加时命中率与加速比有所下降。

5.2 量子遗传算法的变异算子

5.2.1 变异算子形式

因初始几率幅随机，若几率幅分布不如人意，成功搜索（解）能达到的最大概率可能就不那么理想，这是因为对应解的量子态几率幅相对非解量子态的几率幅要低，类似于经典遗传算法中的有效基因缺损，而变异操作增加了物种的多样性，在一定程度上能够改变这个问题。

因此我们在 Grover 迭代的过程中引入变异算子，尝试在种群演化过程中改变解向量的几率幅从而增大得解概率。由于和经典遗传算法类似，量子遗传算法里也是使用量子比特进行基因的表示，所以经典遗传算法里变异操作在量子遗传算法里也基本相同，可以采用均匀变异的方式对某个或某些量子比特利用非门进行反转操作，从而达到变异的效果。一种策略是设立阈值P_Δ，在生成随机初始态后计算其得解概率P_i，若$P_i < P_\Delta$，则引入变异算子：对于n位的量子比特，每次迭代中随机生成长度为$l \in [1,n]$基因段进行变异，该基因段上记录上记录着l个离散的比特位，每个比特位都有独立的变异概率p，仅当变异概率p大于预先设立的变异概率阈值p_Δ时，该比特位才利用非门进行反转，达到变异的效果。

5.2.2 算法实现

Grover推广算法变异操作过程
1. 随机生成长度为$L(L \le n)$的数组$M_i = k(L \le n)$，其中随机数k代表进行变异的量子比特位 2. 生成长度为L的概率数组$P_i = t(0 < t < 100)$，其中随机数t代表每次变异的变异概率 3. 在i次Grover迭代的t(i)时刻，遍历数组M，若相应的P_i值超过预先设立的阈值P_Δ，则对相应量子比特位进行翻转，即进行$X(M_i)$操作

（算法 5.3）

在引入了变异操作之后，上文中出现的随机初始态 Grover 推广算法可以进行进一步改进，算法过程如下：

带变异算子的随机初始态的Grover推广算法迭代过程
1. 根据算法5.1，得到随机初始态$\|\psi_{ran}>$ 2. 根据随机初始态$\|\psi_{ran}>$计算搜索成功率，若成功率低于预先设立的阈值，设立变异标志位$Flag_m$=1，即引入变异算子的标志 3. 通过$O\left(\sqrt{N/t}\right)$次迭代完成搜索过程 3.1 根据$Flag_m$是否为1，判断是否执行算法5.3中的变异操作 3.1 应用oracle算子检验每个元素是否为搜索问题的解 3.2 应用Hadamard变换$H^{\otimes n}$ 3.3 在计算机上执行条件相移，使|0>以外的每个计算基态获得-1的相位移动 $$\|x>\rightarrow -(-1)^{\delta_{x0}}\|x>$$ 3.4 应用Hadamard变换$H^{\otimes n}$

（算法 5.4 见附录 C）

5.2.3 算法表现

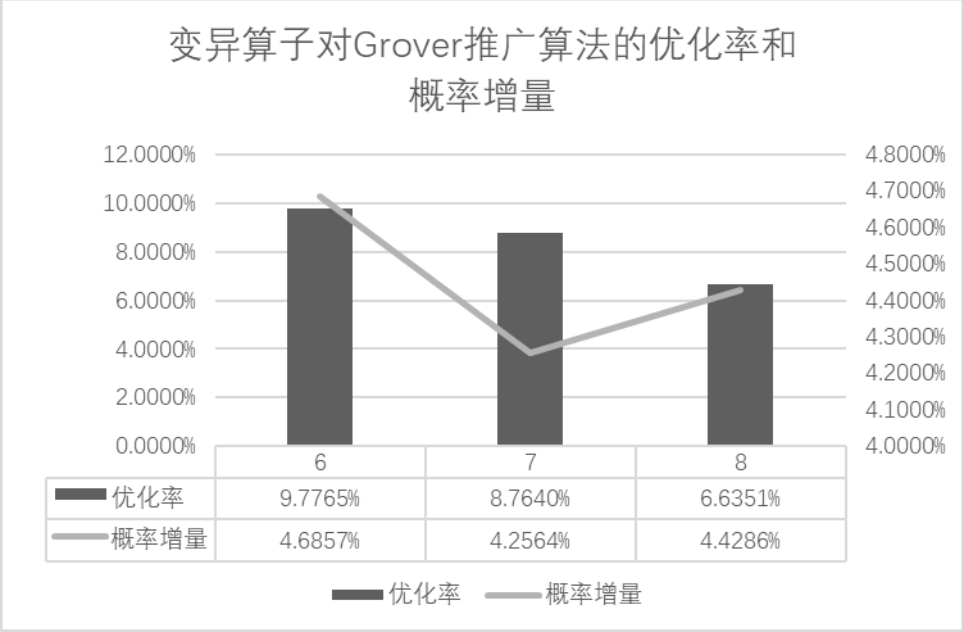


表 5-2 变异算子对 Grover 推广算法的优化率和概率增量

量子遗传算法中的变异操作与经典无太大差异，区别只是作用于量子位或是经典比特。由上表可知，将变异算子引入 Grover 推广算法能够带来一定的算法效率优化和得解概率增量；其中表现最好的为量子比特数为 6 时，能够以接近 10%的概率给 Grover 推广算法带来 4.7%左右的概率增量。

在变异过程中，参数的设置对变异算子的表现有着一定的影响，当量子比特逐渐增加时，变异算子的平均表现在下降，需要通过调整变异算子中的参数如变异段长度、变异概率进行调整，使得在多量子比特时变异算子在 Grover 推广算法中仍能发挥一定的作用。

6 结论

1.主要工作

本文首先将经典遗传算法和量子Grover搜索算法的一个变形——BBHT最优查找算法应用于 0-1 背包问题中进行对比，展示了量子机制给算法带来的高加速比优势。之后，在原始Grover算法的基础上引入（随机生成的）任意初始态与变异算子的概念，探究实现Grover算法的推广形式，并通过实验研究算法的表现情况。

主要工作如下：

（1）本文复现了重要的量子算法——Grover算法和基于该算法的BBHT算法，通过研究量子计算原理、利用近期推出的Microsoft Q#编程语言及其相应组件让算法得以在Microsoft Visual Studio 2017 平台上运行.

（2）通过将经典遗传算法与BBHT算法应用0-1背包问题，研究在相同条件下经典算法与量子算法在命中率和收敛速度的表现。

（3）探究实现了 Grover 算法在初态为非均匀态时的推广形式。采用一定策略随机产生若干个体形成初始叠加态，并采取重复实验的方法分析Grover推广算法相比原始Grover算法的搜索命中率和加速比（迭代次数比值）。

（4）在实现了任意初态的Grover推广算法后，引入变异算子，在Grover迭代的过程中对那些搜索表现不佳的种群（叠加态）以一定的预设概率进行变异，并通过重复实验分析变异操作给 Grover 推广算法带来

的搜索效率（如命中率及收敛速度）的变化。研究发现在低量子比特数时，变异算子能够发挥一定的算法优化作用，但随着量子比特数增加这种优化作用在逐渐衰弱。

2.存在的问题

（1）量子编程语言 Q#实际上实现的是对量子计算机制的模拟，尽管对经典机制和量子机制进行了分离，其本身还是运行在经典计算平台，而用经典计算机模拟 Grover 迭代进程是需要耗费大量计算机资源的，并且耗费量会随着工作的量子比特的增加或是数据库规模的扩大而急剧增长，致使本文中实验采取量子比特均不超过十个。在量子计算机面世后，算法的运行效率会有质的飞跃，但对目前的编程语言来说，鉴于计算机的有限资源，能够模拟的量子比特数不足是一大问题。

（2）在 Grover 推广算法中，任意初始态的随机选择策略和随机变异的策略对最终算法的表现都有较大的影响。由于时间有限，本文只研究了一般性的随机策略，并未加入启发，也没有进行更全面及充分的实验探究参数设置或是变异时机等，致使实验数据没能达到完美的预期。

3.未来展望

本文的拓展工作可以有以下几个方面：①优化初始态随机生成策略，使其在多量子比特（多于十个）时仍能够随机挑选量子比特使其处于纠缠态中；②进行多次实验，判断在一定量子比特数和一定数据规模的标记元素下的最佳变异概率参数，同时除了均匀变异外可以多采取其他变异策略，探究变异策略对 Grover 推广算法表现的影响；③尝试将 Grover 推广算法

应用于 0-1 背包问题中并和经典遗传算法与 BBHT 算法进行算法表现上的对比分析等。

致谢

在此我想衷心感谢我的导师余芳老师，本论文的完成离不开她对我的帮助与支持，大二大三有幸能够参加老师的课，被她严谨的上课模式所折服。但是由于我本身的原因致使项目拖延了很长一段时间，心里觉得很对不起老师；再次拾起项目已是四月初，但是余芳老师仍然用她的严谨的治学态度和极大的耐心帮助我完成实验以及论文的编写。也希望自己在今后的道路上尽心尽力，不要留下遗憾。

感谢大学四年中我遇到的所有任课老师以及辅导员邓老师、潘老师，从各位老师身上不仅学习到了很多专业知识，还学习到了许多受益终生的人生道理。

感谢外宿生们在我最迷茫时给我的精神慰藉，感谢挚友严志乐的陪伴和教诲。感谢雷彩平同学在生活方面无微不至的扶持，感谢马银敏同学在大学四年中为班集体所付出的一切以及对我的关怀。

感谢父母多年来对我的悉心栽培，以及一直在背后默默支持我的选择，没有他们我就不能在人生的道路上坚持与奋斗。

附录A

经典遗传算法核心代码

```
var Gene = function(code){
    this.code = code;
    this.cost = 0;
    this.ptw = 0;
    this.MaxWeight = 0;
}
Gene.prototype.code = '';
// 价值函数
Gene.prototype.calcCost = function(MaxWeight){
    this.MaxWeight = MaxWeight;
    var WeightArr = this.WeightArr;
    var ValueArr = this.ValueArr;
    var totalWeight = 0;
    var totalValue = 0;
    for( let i = 0 ; i < this.code.length ; i++ ){
        // 1 表示携带这个东西 0 表示不带这个东西
        if( this.code[i] == 1 ){
            totalWeight += WeightArr[i];
            totalValue += ValueArr[i];
        }
    }
    if( totalWeight > this.MaxWeight ){
        this.cost = 1;
    }else{
        this.cost = totalValue;
    }
    this.ptw = totalWeight;
    return this.cost;
}
// 交配函数
Gene.prototype.mate = function(gene){
    var pivot = Math.round( Math.random() * this.Len );
    var child1 = this.code.substr(0 , pivot ) + gene.code.substr(pivot);
    var child2 = gene.code.substr(0 , pivot ) + this.code.substr(pivot);
    return [new Gene(child1) , new Gene(child2)];
}
// 突变函数
Gene.prototype.mutate = function(P_MUTATION){
```

```
        if( Math.random() > P_MUTATION ) return ;
        var newCode = this.code.split('');
        // 随机突变长度
        var changeLength = Math.floor( Math.random() * this.code.length );
        for(var i = 0 ; i < changeLength ; i ++){
            // 随机突变位置
            var index = Math.floor( Math.random() * this.code.length );
            newCode.splice( index , 1 , Math.round( Math.random() ) )
        }
        this.code = newCode.join('');

}
// 生成随机基因
Gene.prototype.random = function(){
        var temp = '';
        for(var i = 0 ; i < this.Len ; i ++ ){
            temp += Math.round( Math.random() );
        }
        this.code = temp;
}
var Population = function(pop){
        this.MaxWeight = pop.bag.maxW; //背包最大承重
        this.WeightArr = pop.bag.wArr;//物品重量
        this.ValueArr = pop.bag.vArr; //物品价值
        this.maxValue = 0;   //储存最大价值
        this.maxGene = [];   //储存最大价值组合
        this.times = 0;   //计算出现最大值的次数
        // 最大值在 100（allTimes）代中都相同。
        this.allTimes = pop.similarGen == undefined ? 100 : pop.similarGen ;
        this.Size = pop.size == undefined ? 20 : pop.size ; //种群规模
        this.P_XOVER = 0.8;   //遗传概率
        this.P_MUTATION = pop.P_MUTATION == undefined ? 0.15 : pop.P_MUTATION ; ; //  变异概率
        this.generationNum = 0;   //  繁衍次数
        var self = this;
        // 这里补充写 Gene 的类，方便把 WeightArr 和 ValueArr 传入
        Gene.prototype.WeightArr = self.WeightArr
        Gene.prototype.ValueArr = self.ValueArr
        Gene.prototype.Len = self.WeightArr.length

        this.MaxGenerations = pop.maxGen == undefined ? '100' : pop.maxGen ; //总进化代数
        this.type = pop.type == undefined ? 'near100' : pop.type ; //终止函数的类型（是当 100 代的最大值相同时终止程序还是繁衍 N 代之后终止程序）
        this.members = []; //  种群所有对象
```

```
        // 生成 Gene
        while( this.Size --){
            var gene = new Gene();
            gene.random();
            this.members.push(gene);
        }
}
// 按照分值进行排序
Population.prototype.sort = function(){
        this.members.sort(function(a,b){
            return b.cost - a.cost;
        })
}
Population.prototype.generation = function(){
        var self = this;
        // 计算分值
        for( let i = 0 ; i < self.members.length ; i ++ ){
            var curValue = self.members[i].calcCost(self.MaxWeight)
            if( curValue > self.maxValue){
                self.maxValue = curValue;
                self.maxGene = self.members[i].code;
                self.times = 0;
            }
        }
        // 根据分值排序
        self.sort();
        // 展示函数，结果即时体现
        self.display();

        // 交配个数
        var mateNum = 3;
        // 产生后代总数
        var newChildNum = mateNum*(mateNum-1)
        // 把种群末尾的淘汰掉，把新后代添加到种群
        self.members.splice( self.members.length - newChildNum , newChildNum )
        for(var i = 0 ; i < mateNum ; i ++){
            for(var j = i+1 ; j < mateNum ; j ++){
                var chi = self.members[i].mate( self.members[j] );
                self.members.push( chi[0] )
                self.members.push( chi[1] )
            }
        }
        // 突变
```

```
        for( var i = 0 ; i < self.members.length ; i ++ ){
            self.members[i].mutate(    self.P_MUTATION );
            var curValue = self.members[i].calcCost(self.MaxWeight)
            if( curValue > self.maxValue){
                self.times = 0;
                self.maxValue = curValue;
                self.maxGene = self.members[i].code;
            }
            // 判断啥时候终止程序。
            if (self.type == 'maxGen' && self.generationNum >= self.MaxGenerations) {
                self.stop();
                return true;
            } else if (self.type == 'near100' && self.times == self.allTimes) { //连续 100 代不变就停止
                self.stop();
                return true;
            }
        }
        self.times++
        self.generationNum ++ ;

        setTimeout( function(){
            self.generation();
        } , 20);
}
// 终止程序
Population.prototype.stop = function(){
        var self = this;
        if(maxmax < self.maxValue){
            maxmax = self.maxValue
        }
        // 辅助画图的。
        option.yAxis.max = 2*maxmax;
        option.series[0].data.push(self.maxValue);
        myChart.setOption(option);
        self.sort();
        self.display();
        perResult.push({maxV:self.maxValue,gene:self.maxGene })
        if(perResult.length == loopTimes){
            self.showTable(perResult)
        }
}
// 展现表格结果
Population.prototype.showTable = function(perResult){
```

```
    var string = '';
    var table = document.getElementById('table');
    for(var i = 0 ; i < perResult.length ; i ++ ){
        if(perResult[i].maxV == maxmax){
            string += '<tr class="info"><th scope="row">'+ (i+1) +'</th><td>'+ perResult[i].gene +'</td><td>'+ perResult[i].maxV +'</td></tr>';
        }else{
            string += '<tr><th scope="row">'+ (i+1) +'</th><td>'+ perResult[i].gene +'</td><td>'+ perResult[i].maxV +'</td></tr>';
        }
    }
    table.innerHTML += string
}
// 展现动态的繁衍结果
Population.prototype.display = function(){
    var result = document.getElementById('result');
    result.innerHTML = '';
    result.innerHTML += ("<h2>Generation: " + this.generationNum + "</h2>");
    result.innerHTML += ("<ul>");
    for (var i = 0; i < this.members.length; i++) {
        result.innerHTML += ("<li>" + this.members[i].code + " (" + this.members[i].cost + ")");
    }
    result.innerHTML += ("</ul>");
}
// 入口配置
function fun (pop) {
    window.myChart = echarts.init(document.getElementById('main'));
    // 指定图表的配置项和数据
    window.loopTimes = pop.loopTimes == undefined ? 10 : pop.loopTimes;
    window.maxmax = 0;
    window.perResult = [];
    window.option = {
        title: {
            text: '循环'+ loopTimes +'次得分情况'
        },
        tooltip: {},
        legend: {
            data: ['value']
        },
        xAxis: {
            data: []
        },
        yAxis: {
```

```
            type: 'value',
            max: 60
        },
        series: [{
            name: 'value',
            type: 'line',
            data: [],
        }],
        label: {
            normal: {
                show: true
            }
        }
    };
    for (var i = 0; i < loopTimes; i++) {
        option.xAxis.data.push(i+1);
        var population = new Population(pop);
        population.generation()
    }
}
window.onload = function(){
    // 配置项
    var pop = {
        // type:'maxGen',   //终止函数类型 ，  默认 near100
        // maxGen:5,          //繁衍后代数（在 type 为'maxGen'时有用），默认 100
        // bag:{
        //     maxW:150,       //背包限重
        //     wArr : [35,30,60,50,40,10,25],   // 物品重量
        //     vArr : [10,40,30,50,35,40,30]    // 物品价值
        // },
        bag:{
            maxW:30,
            wArr : [3,5,7,10,15,20,25,6,8,2],
            vArr : [10,6,5,8,4,6,12,3,2,1]
        },
        loopTimes:10,    //循环次数（可以理解为实验次数）， 默认 10
        // P_MUTATION:0.15,    //变异概率 ，  默认 0.15
        similarGen:20,    //决定多少个后代的最大值相同时停止程序。（在 type 为 near100 时有用），
默认 100
        // size:30      //种群大小(建议 20~100)，默认 20
    }
    fun(pop);
}
```

附录B

BBHT 算法核心源代码

```
/// Implements the operation |x> → (-1)^f(x) |x>
/// where f(x) = 1 when |x> = |11…1> and 0 otherwise
operation DatabaseOracleFromInts(markedElements : Int[], markedQubit: Qubit, databaseRegister: Qubit[]) : Unit
{
    body (...) {
        let nMarked = Length(markedElements);
        for (idxMarked in 0..nMarked - 1) {
            // Note: As X accepts a Qubit, and ControlledOnInt only
            // accepts Qubit[], we use ApplyToEachCA(X, _) which accepts
            // Qubit[] even though the target is only 1 Qubit.
            (ControlledOnInt(markedElements[idxMarked], ApplyToEachCA(X, _)))(databaseRegister, [markedQubit]);
        }

    }
    adjoint auto;
    controlled auto;
    adjoint controlled auto;
}

/// # Summary
/// This implements an oracle `DU` that prepares the start state
/// DU|0⟩|0⟩ = √(M/N)|1⟩|marked⟩ + √(1-(M/N)^2)|0⟩|unmarked⟩ where
/// `M` is the length of `markedElements`, and
/// `N` is 2^n, where `n` is the number of database qubits.
operation GroverStatePrepOracleImpl(markedElements : Int[], idxMarkedQubit: Int , startQubits: Qubit[]) : Unit
{
    body (...) {
        let flagQubit = startQubits[idxMarkedQubit];
        let databaseRegister = Exclude([idxMarkedQubit], startQubits);

        // Apply oracle `U`
        ApplyToEachCA(H, databaseRegister);

        // Apply oracle `D`
        DatabaseOracleFromInts(markedElements, flagQubit, databaseRegister);
```

```
        }

        adjoint auto;
        controlled auto;
        adjoint controlled auto;
    }

    /// # Summary
    /// `StateOracle` type for the preparation of a start state that has a
    /// marked qubit entangled with some desired state in the database
    /// register.
    function GroverStatePrepOracle(markedElements : Int[]) : StateOracle
    {
        return StateOracle(GroverStatePrepOracleImpl(markedElements, _, _));
    }

    /// # Summary
    /// On input |0⟩|0⟩, this prepares the state |1⟩|marked⟩ with amplitude
    /// Sin((2*nIterations + 1) ArcSin(Sqrt(M/N))).
    function GroverSearch( markedElements: Int[], nIterations: Int, idxMarkedQubit: Int) : (Qubit[] =>
Unit : Adjoint, Controlled)
    {
        return AmpAmpByOracle(nIterations, GroverStatePrepOracle(markedElements),
idxMarkedQubit);
    }

    // Let us now allocate qubits and run GroverSearch.

    /// # Summary
    /// Measurement outcome of marked Qubit and measurement outcomes of
    /// the database register converted to an integer.
    operation ApplyGroverSearch( markedElements: Int[], nIterations : Int, nDatabaseQubits : Int) :
(Result,Int) {
        body (...) {
            // Allocate variables to store measurement results.
            mutable resultSuccess = Zero;
            mutable numberElement = 0;

            // Allocate nDatabaseQubits + 1 qubits. These are all in the |0⟩
            // state.
            using (qubits = Qubit[nDatabaseQubits+1]) {
```

```
            // Define marked qubit to be indexed by 0.
            let markedQubit = qubits[0];

            // Let all other qubits be the database register.
            let databaseRegister = qubits[1..nDatabaseQubits];

            // Implement the quantum search algorithm.
            (GroverSearch( markedElements, nIterations, 0))(qubits);

            // Measure the marked qubit. On success, this should be One.
            set resultSuccess = M(markedQubit);

            // Measure the state of the database register post-selected on
            // the state of the marked qubit.
            let resultElement = MultiM(databaseRegister);

            set numberElement = PositiveIntFromResultArr(resultElement);

            // These reset all qubits to the |0⟩ state, which is required
            // before deallocation.
            ResetAll(qubits);
        }

        // Returns the measurement results of the algorithm.
        return (resultSuccess, numberElement);
    }
}
```

附录C

随机初始态 Grover 推广算法及变异算子核心源代码

```
operation MutationPreparation(nQubits: Int) : (Int, Int[], Int[]){
        //First random number is for mutation length
        let ranNum = RandomInt(nQubits);
        //Second random number array is for mutation position
        mutable ranArray = new Int[ranNum];
        //Mutation Posibility controls the posilibity that each mutation performs
        //At the range of 0..99, integers under 15 allows mutation to perform, indicating that the overall
        //mutation posibility is 15%
        mutable mutationPosibility = new Int[ranNum];
        for(idxRan in 0..ranNum-1)
        {
            set ranArray w/= idxRan <- RandomInt(nQubits);
            set mutationPosibility w/= idxRan <- RandomInt(100);
        }
        return (ranNum, ranArray, mutationPosibility);
    }

    /// Implements the operation |x> → (-1)^f(x) |x>
    /// where f(x) = 1 when |x> = |11…1> and 0 otherwise
    operation DatabaseOracle ( markedElements: Int[], markedQubit : Qubit, databaseRegister : Qubit[]) : Unit is Adj + Ctl{
        body(...){
            // Equivalent to CNOT(databaseRegister, markedQubit)
            // The Controlled functor apply its function conditioned on the state |11…1>

            // A sequence of integers are marked
            let nMarked = Length(markedElements);
            for (idxMarked in 0..nMarked - 1) {
                // Note: As X accepts a Qubit, and ControlledOnInt only
                // accepts Qubit[], we use ApplyToEachCA(X, _) which accepts
                // Qubit[] even though the target is only 1 Qubit.
                (ControlledOnInt(markedElements[idxMarked], ApplyToEachCA(X, _)))(databaseRegister, [markedQubit]);
            }
        }

        //adjoint auto;
```

```
    }

    /// # Summary
    /// To perform the uniform superposition on every single Qubit,
    /// and the register |s> will be |++…+> eventually
    /// # Input
    /// ## databaseRegister
    /// Qubits all in |0> state
    operation UniformSuperpositionOracle(databaseRegister: Qubit[], initialSign: Int, HArray : Int[], XArray : Int[], CNOTLeft : Int[], CNOTRight : Int[]) : Unit is Adj + Ctl{
        body(...){
            let nQubits= Length(databaseRegister);

            let HLength = Length(HArray);
            let XLength = Length(XArray);
            let CNOTLength = Length(CNOTLeft);
            /// INITIAL STATE

            ApplyToEachCA(H, databaseRegister);
            if (initialSign == 1) {
                //Random flip
                for (idxQubit in 0..XLength-1) {
                    X(databaseRegister[XArray[idxQubit]]);
                }
                //Random Controlled flip
                for (idxQubit in 0..CNOTLength-1) {
                    if(databaseRegister[CNOTLeft[idxQubit]] != databaseRegister[CNOTRight[idxQubit]]){
                        CNOT(databaseRegister[CNOTLeft[idxQubit]], databaseRegister[CNOTRight[idxQubit]]);
                    }
                }
                //Random Hadamard transformation
                for (idxQubit in 0..HLength-1) {
                    H(databaseRegister[HArray[idxQubit]]);
                }
            }

        }

    }

    /// #Summary
```

```
    /// Given a register of qubits initially in the state |00…0>, prepares the start state
    /// as |1⟩|N-1⟩/ √ N + |0⟩(|0⟩+|1⟩+...+|N-2⟩)/ √ N.
    operation StatePreparationOracle( markedElements: Int[], markedQubit: Qubit, databaseRegister: Qubit[],
                                                ranLength: Int, ranArray: Int[], posibility: Int[], mutation: Int, initialSign: Int, HArray: Int[], XArray: Int[], CNOTLeft: Int[], CNOTRight: Int[]) : Unit is Adj{
        body(...){
            UniformSuperpositionOracle(databaseRegister, initialSign, HArray, XArray, CNOTLeft, CNOTRight);

            /// MUTATION may be performed here
            /// NOTE
            for(idxRan in 0..ranLength-1)
            {
                if(posibility[idxRan] < 30 and mutation == 1)
                {
                    X(databaseRegister[ranArray[idxRan]]);
                    //Message($"Mutation position at: {ranArray[idxRan]}");
                }
            }

            DatabaseOracle(markedElements, markedQubit, databaseRegister);
            //Message($"Mutation length is: {ranLength}");

        }
    }

    /// #Summary
    /// The Reflection about the marked state
    operation ReflectMarked(markedQubit : Qubit) : Unit{
        body(...){
            //Marked elements always have the marked qubit in the state |1>
            R1(PI(), markedQubit);
        }
    }

    /// #Summary
    /// The Reflection about the |00…0⟩ state
    operation ReflectZero(databaseRegister : Qubit[]) : Unit{
        ApplyToEachCA(X, databaseRegister);
        Controlled Z(Rest(databaseRegister), Head(databaseRegister));
        ApplyToEachCA(X, databaseRegister);
    }
```

```
/// #Summary
/// The Reflection about the start state
/// The pseudocode can be written as follows
/// 1. Apply H gates to every qubit
/// 2. Apply X gates to every qubit
/// 3. Apply an n-1 controlled Z gate to the 0th qubit
/// 4. Apply X gates to every qubit
/// 5. Apply H gates to every qubit
/// which is also the algorithm described in "Quantum computation and Quantum
/// information(Chinese Edition)" P231
operation ReflectStart( markedElements: Int[], markedQubit: Qubit, databaseRegister: Qubit[], mutation: Int) : Unit{
    body(...){
        ///Preparation for mutations
        let nQubits = Length(databaseRegister);
        mutable ranNum_I = RandomInt(nQubits);
        mutable ranArray_I = new Int[ranNum_I];
        mutable mutationPosibility_I = new Int[ranNum_I];
        set (ranNum_I, ranArray_I, mutationPosibility_I) = MutationPreparation(nQubits);

        (Adjoint StatePreparationOracle)(markedElements, markedQubit, databaseRegister, ranNum_I, ranArray_I, mutationPosibility_I, mutation, 0, [0], [0], [0], [0]);
        ReflectZero([markedQubit] + databaseRegister);

        ///Preparation for mutations
        mutable ranNum_II = RandomInt(nQubits);
        mutable ranArray_II = new Int[ranNum_II];
        mutable mutationPosibility_II = new Int[ranNum_II];
        set (ranNum_II, ranArray_II, mutationPosibility_II) = MutationPreparation(nQubits);

        StatePreparationOracle(markedElements, markedQubit, databaseRegister, ranNum_II, ranArray_II, mutationPosibility_II, mutation, 0, [0], [0], [0], [0]);
    }
}

/// RS      : ReflectedStart
/// RM      : ReflectedMarked
/// M       : Iteration times
/// |s>    : Start state
/// (RS · RM)^M |s⟩ = sin((2M+1)θ) |1⟩|N-1⟩
///                          + cos((2M+1)θ) |0⟩(|0⟩+|1⟩+...+|N-2⟩)
/// The product RS · RM is known as the Grover iterator, and each application
```

```
/// of it rotates |s⟩ in the two-dimensional subspace by angle 2θ.
operation QuantumSearch( markedElements: Int[], nIterations: Int, markedQubit: Qubit, databaseRegister: Qubit[],
                                HArray: Int[], XArray: Int[], CNOTLeft: Int[], CNOTRight: Int[], mutation: Int) : Unit{
    body(...){
        ///Preparation for mutations
        let nQubits = Length(databaseRegister);
        mutable ranNum = RandomInt(nQubits);
        mutable ranArray = new Int[ranNum];
        mutable mutationPosibility = new Int[ranNum];
        set (ranNum, ranArray, mutationPosibility) = MutationPreparation(nQubits);

        let initialSign = 1;
        StatePreparationOracle(markedElements, markedQubit, databaseRegister, ranNum, ranArray, mutationPosibility, mutation, initialSign, HArray, XArray, CNOTLeft, CNOTRight);
        //Loop over Grover iterates
        for(idx in 0..nIterations-1){
            ReflectMarked(markedQubit);
            ReflectStart(markedElements, markedQubit, databaseRegister, mutation);
        }
    }
}

/// # Summary
/// Performs quantum search for the marked element and returns an index
/// to the found element in binary format. Finds the marked element with
/// probability sin²((2*nIterations+1) sin⁻¹(1/ √ N)).
/// ** θ = sin⁻¹(1/ √ N) that is because the grover state rotates π/2 in total
///        to be in the solution state, so N² = sin(π/2) / sin(θ)
operation ApplyQuantumSearch( markedElements: Int[], nIterations : Int, nDatabaseQubits : Int,
                                HArray : Int[], XArray : Int[], CNOTLeft : Int[], CNOTRight : Int[], mutation : Int) : (Result, Int){
    body(...){
        //Variables to store measurement results
        mutable resultSuccess = Zero;
        mutable resultElement = new Result[nDatabaseQubits];
        mutable numberElement = 0;

        using(qubits = Qubit[nDatabaseQubits+1]){
            let markedQubit = qubits[0];
            let databaseRegister = qubits[1..nDatabaseQubits];
```

```
                QuantumSearch( markedElements, nIterations, markedQubit, databaseRegister, HArray,
XArray, CNOTLeft, CNOTRight, mutation);

                set resultSuccess = M(markedQubit);
                set resultElement = MultiM(databaseRegister);

                set numberElement = ResultArrayAsInt(resultElement);

                //Reset all the qubits to the |0> state
                //which is required before deallocation
                if(resultSuccess == One){
                    X(markedQubit);
                }
                for(idxResult in 0..nDatabaseQubits-1){
                    if(resultElement[idxResult] == One){
                        X(databaseRegister[idxResult]);
                    }
                }
            }
            // Returns the measurement results of the algorithm
            return(resultSuccess, numberElement);
        }
    }
```

参考文献

[1] Darrell Whitley. A genetic algorithm tutorial[J]. Statistics and Computing, 1994, 4: 65-85

[2] D E Goldberg. Genetic algorithms in search, optimization, and machine learning[B]. MA: Addison-Wesley, 1989..

[3] C Aporntewan, P Chongstitvatana. A hardware implementation of the compact genetic algorithm[E]. In:. Proceedings of the 2001 Congress on Evolutionary Computation. Korea:. 2001. 624－629.

[4] 王小平,曹立明. 遗传算法—理论、应用与软件实现[M]. 西安. 西安交通大学出版社, 2002

[5] 李岩,袁弘宇,于佳乔,张更伟,刘克平. 遗传算法在优化问题中的应用综述[J]. 山东工业技术, 2019(12):242-243+180.

[6] P W Shor. Algorithms for quantum computation: Discrete algorithms and factoring[E]. In:. Proceedings 35th Annual Symposium on Foundations of Computer Science. USA:. Nov 1994. 124－134.

[7] L K Grover. Quantum mechanics helps in searching for a needle in a haystack[J]. Phys. Rev. Lett., 1997, 79(2): 325－328

[8] 杨淑媛. 量子进化算法的研究及其应用[D]. 西安: 西安电子科技大学, 2003.

[9] 钱洁, 王保华, 郑建国, 等. 多重二次背包问题的量子进化求解算法[J]. 计算机学报, 2015, 38(8): 1518－1529.

[10] Wang H, Li L, Liu J, Wang Y, Fu C. Improved quantum genetic algorithm in application of scheduling engineering personnel [J]. Appl. Anal., 2014, 1–10.

[11] Manju A, Nigam M J. Applications of quantum inspired computational intelligence: a survey[J]. Artificial Intelligence Review, 2014, 42(1): 79–156.

[12] Rylander B, Soule T, Foster J, and Alves-Foss J. Quantum evolutionary programming [E]. In:. Proc. Genetic Evol. Comput. Conf. (GECCO-2001). California:. 2001. 1005–1011.

[13] Udrescu M, Prodan L, Vlăduţiu M. Grover’s algorithm and the evolutionary approach of quantum computation[K]. ACSA Report. România: Politehnica University of Timisoara, 2004. http://www.acsa.upt.ro/publications/index.htm

[14] Udrescu M, Prodan L, Vlăduţiu M. Implementing quantum genetic algorithms: A solution based on Grover’s algorithm [E].. In:. Proceedings of the 3rd Conference on Computing Frontiers. Ischia, Italy:. 2006. 71–81.

[15] Eli B, Ofer B, David B, Markus G, and Daniel A L. Grover’s quantum search algorithm for an arbitrary initial amplitude distribution[J]. PHYSICAL REVIEW A, 1999, 60(4):2742-2745

[16] 葛继科, 邱玉辉, 吴春明等. 遗传算法研究综述[J]. 计算机应用研

究, 2008, 25(10): 2912-2914.

[17] 张旅兰. 混合遗传算法在布尔方程求解中的应用[D]. 北京: 北京工商大学, 2006.

[18] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to algorithm[M]. Cambridge: The MIT Press, 2001.

[19] 赵学武,刘向娇,王兴,刘兵杰.求解 0-1 背包问题的遗传算法[J].南阳师范学院学报,2014,13(06):21-25.

[20] 李岩, 袁弘宇, 于佳乔, 张更伟, 刘克平. 遗传算法在优化问题中的应用综述[J]. 山东工业技术, 2019(12): 242-243+180.

[21] A Peres. Quantum Theory: Concepts and Methods[B]. Norwell, MA: Kluwer, 1998.

[22] 蔡照鹏, 杨盛苑. 基于遗传算法的 0/1 背包问题的改进算法[J]. 河南城建学院学报, 2013, 22(04): 60-62+68.

[23] Andrea M, Enrico B, Tommaso C. Quantum Genetic Optimization[J]. IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, 2008, 12(2): 231-241

[24] M Boyer, G Brassard, P Hoyer, A Tapp. Tight bounds on quantum searching[J]. Fortschr Phys, 1998, 46:493