量子搜索算法的课外探究

1st 付容天	2nd 王子晗	3rd 吴清柳	4th 汤润清
SCS, BUPT	SCS, BUPT	SCS, BUPT	SCS, BUPT
学号 2020211616	学号 2020211596	学号 2020211597	学号 2020211595
班级 2020211310	班级 2020211310	班级 2020211323	班级 2020211310

Abstract—**在学习了量子搜索算法之后，我们对 Grover 算法的诸多细节与应用产生了兴趣。在本文中，我们探究了课上没有覆盖的 Grover 算法的一些细节、学习了 Grover 算法的一些优化技巧。我们还使用 Grover 算法解决了图着色、整数分解和旅行商三个问题，其中图着色问题还进行了代码实现。**

Index Terms—**量子搜索, 相位估计, 图着色问题, 整数分解问题, 旅行商问题**

I. 量子搜索算法回顾

在现实生活中，往往存在这样一种问题：给定一个点的集合，其中每个点之间可能有边连接，要求求出从指定起点到终点的一条最短路径。这个问题的一般化表述为：在大小为 N 的缺少结构化信息的搜索空间中，希望找到 M 个满足已知性质的目标元素。通常来讲，该算法的时间复杂度与元素的数量成正比，也就是说，在经典计算机上，必须遍历所元素，同时逐一观察，最后才能找出满足一致性质的那个元素，即时间复杂度为 $O(N/M)$。但是在量子搜索算法（Grover 算法）中，我们可以利用量子性质实现加速，此时搜索的时间复杂度仅为 $O(\sqrt{N/M})$。

在经典计算中，如果要判断一个元素是否满足目标，往往可以将此问题抽象为：选择合适的方式将输入映射为 0 或 1。也就是说，对于经典计算机而言，如果有 n 比特输入 $x \in \{0,1\}^n$，那么目标就是选择出一个合理的 $f(x)$ 完成 $\{0,1\}^n \to \{0,1\}$ 这一映射过程，从而可以得知当前元素是否为满足目标性质的元素。这种“一一检验”的思路很容易实现，但却是制约时间复杂度的瓶颈——必须逐一检验所有元素才能得到结果。

因此，为了加速搜索过程，很自然地想到了量子的并行性，并行性能否作用于搜索过程？Lov K. Grover 提出了一个有效的量子搜索算法 [1]，现在来简要介绍这一算法。

算法的思路是在解空间中标记出想要的那些解，因此，算法的第一步就是构造解空间。n 个初始状态 $|0\rangle$ 通过 H 门，可以得到量子态：

$$|\phi\rangle = \frac{\sum_{x_1,x_2,\cdots,x_n=0,1} |x_1x_2\cdots x_n\rangle}{(\sqrt{2})^n}$$

换句话说，我们得到了 n 量子比特的一个均匀叠加，n 量子比特所能表示的所有状态都均匀叠加到了状态 $|\phi\rangle$ 中。接下来需要考虑的就是如何在状态 $|\phi\rangle$ 中标记出想要的那些状态（所对应的分量）。

首先需要定义：什么是想要的状态、以及它（们）有什么性质？我们想要的状态是满足指定性质的状态，可以定义一个 $\{0,1\}^n \to \{0,1\}$ 的映射 $f(x)$，$f(x)$ 的二元输出自然对应了满足与不满足指定性质两种情况。接下来，我们将状态 $|\phi\rangle$ 按照其分量是否满足指定性质分成两部分：

$$\begin{aligned}|\phi\rangle =& \frac{\sum_{f(|x_1x_2\cdots x_n\rangle)=0} |x_1x_2\cdots x_n\rangle}{(\sqrt{2})^n} \\ &+ \frac{\sum_{f(|x_1x_2\cdots x_n\rangle)=1} |x_1x_2\cdots x_n\rangle}{(\sqrt{2})^n}\end{aligned}$$

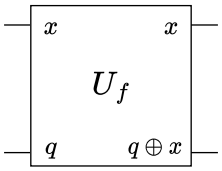
记 $(x)_{10} = (x_1x_2\cdots x_n)_2$，即 x 为二进制数 $x_1x_2\cdots x_n$ 对应的十进制数，且记 $N = 2^n$，分别设上式中前一部分有 α 项、后一部分有 β 项（$\alpha + \beta = 2^n$），那么：

$$\begin{aligned}|\phi\rangle &= \sqrt{\frac{\alpha}{N}} \frac{\sum_{f(|x\rangle)=0} |x\rangle}{\sqrt{\alpha}} + \sqrt{\frac{\beta}{N}} \frac{\sum_{f(|x\rangle)=1} |x\rangle}{\sqrt{\beta}} \\ &= \cos\theta|\phi_0\rangle + \sin\theta|\phi_1\rangle\end{aligned}$$

下面考虑将这个状态集中到 $|\phi_1\rangle$ 上，即考虑增加其振幅值 $\sin\theta$。我们前面已经提到：想要的目标解是满足

$f(|x\rangle) = 1$ 所对应的那些 $|x\rangle$，那么现在来考虑如何利用这个函数 f 来进行幅值放大。

假设按照下图所示的思路构造出 U_f 门：



那么对于输入 $|x\rangle$ 和 $|q\rangle$，可以得到：

$$|x\rangle|q\rangle \longrightarrow |x\rangle|q \oplus f(x)\rangle$$

其中，$|q\rangle$ 为单量子比特，$\oplus$ 表示模 2 加。如果将第二个输入量子比特设为

$$|q\rangle = \frac{(|0\rangle - |1\rangle)}{\sqrt{2}}$$

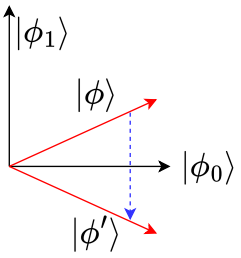
那么对于输入态 $|x\rangle|q\rangle$，有：

$$|x\rangle\left(\frac{(|0\rangle - |1\rangle)}{\sqrt{2}}\right) \longrightarrow (-1)^{f(x)}|x\rangle\left(\frac{(|0\rangle - |1\rangle)}{\sqrt{2}}\right)$$

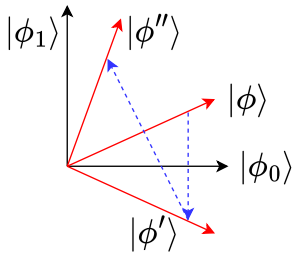
这是因为

$$\begin{aligned} U_f|x\rangle(|0\rangle - |1\rangle) &= U_f|x\rangle|0\rangle - U_f|x\rangle|1\rangle \\ &= |x\rangle|0 \oplus f(x)\rangle - |x\rangle|1 \oplus f(x)\rangle \\ &= \begin{cases} |x\rangle|0\rangle - |x\rangle|1\rangle, & f(x) = 0 \\ |x\rangle|1\rangle - |x\rangle|0\rangle, & f(x) = 1 \end{cases} \\ &= (-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle) \end{aligned}$$

因此，如果令 $|\phi\rangle$ 通过该 U_f 门，则会得到 $|\phi'\rangle = \cos\theta|\phi_0\rangle - \sin\theta|\phi_1\rangle$，在几何上，这相当于下面的变换：



之前我们提到，$|\phi_1\rangle$ 是所有满足 $f(|x\rangle) = 1$ 的 $|x\rangle$ 的叠加态。现在看来，经过 U_f 门之后的 $|\phi\rangle$ 似乎偏离了我们的目标，这时我们不妨考虑将 $|\phi'\rangle$ 做关于 $|\phi\rangle$ 的轴对称操作，即下图所示的操作，该操作可以让 $|\phi\rangle$ 更加偏向 $|\phi_1\rangle$：



在进一步讨论之前，我们必须明确一个事实：起始状态 $|\phi\rangle$ 通常接近所有不满足 $f(|x\rangle) = 1$ 的 $|x\rangle$ 的叠加状态 $|\phi_0\rangle$、同时通常远离所有满足 $f(|x\rangle) = 1$ 的 $|x\rangle$ 的叠加状态 $|\phi_1\rangle$，这是因为往往在解空间中只有少数向量是符合我们要求的解向量，因此其构成的 θ 将会非常小，故 θ 通常满足 $0 \le \theta \le \pi/4$。

下面进行进一步的讨论。在之前的两步中，第一步对 $|\phi\rangle$ 关于 $|\phi_0\rangle$ 做轴对称得到 $|\phi'\rangle$，第二步对 $|\phi'\rangle$ 关于 $|\phi\rangle$ 做轴对称得到 $|\phi''\rangle$，并且 $|\phi''\rangle$ 比 $|\phi\rangle$ 更接近期望状态 $|\phi_1\rangle$。如果选定 $\{|\phi_0\rangle, |\phi_1\rangle\}$ 基，则有：

$$\begin{aligned} |\phi''\rangle &= (2|\phi\rangle\langle\phi| - I)|\phi'\rangle \\ &= (2|\phi\rangle\langle\phi| - I)(2|\phi_0\rangle\langle\phi_0| - I)|\phi\rangle \\ &= \begin{bmatrix} \cos 2\theta & \sin 2\theta \\ \sin 2\theta & -\cos 2\theta \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}|\phi\rangle \\ &= \begin{bmatrix} \cos 2\theta & -\sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix}|\phi\rangle \end{aligned}$$

记

$$G = \begin{bmatrix} \cos 2\theta & -\sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix}$$

其中 $0 \le \theta \le \pi/4$。G 矩阵就是 Grover 迭代的代数表达，将 $|\phi\rangle$ 经过若干次 Grover 迭代，就可以得到足够接近 $|\phi_1\rangle$ 的一个状态 $G^k|\phi\rangle$。现在的问题就是：应当进行多少次迭代、即 k 应为多少？注意到系统的初始状态 $|\phi\rangle = \sqrt{\alpha/N}|\phi_0\rangle + \sqrt{\beta/N}|\phi_1\rangle$，记 $CI(x)$ 为最接近实数 x 的整数，并按照惯例向下取整，从而得到 Grover 迭代的重复次数为

$$R = CI\left(\frac{\arccos\sqrt{\beta/N}}{2\theta}\right)$$

也就是说，在 α 远大于 β 的前提下，我们可以通过 R 次 Grover 迭代将 $|\phi\rangle$ 变换为足够接近 $|\phi_1\rangle$ 的状态 $G^R|\phi\rangle$。

对于迭代次数，我们还有另一种思考方向，首先容易得到：

$$G^k = \begin{bmatrix} \cos 2k\theta & -\sin 2k\theta \\ \sin 2k\theta & \cos 2k\theta \end{bmatrix}$$

故有：

$$G^k|\phi\rangle = \begin{bmatrix} \cos(2k+1)\theta \\ \sin(2k+1)\theta \end{bmatrix}$$

那么在进行量子测量的时候，测得 $|\phi_1\rangle$ 的概率为 $\sin^2(2k+1)\theta$，因此为了尽可能将此概率接近 1，可得迭代次数为：

$$R = CI\left(\frac{\pi}{4\theta} - \frac{1}{2}\right) \approx CI\left(\frac{\pi}{4}\sqrt{\frac{\beta}{N}}\right)$$

这里，我们可以很明显地看出量子搜索算法对经典搜索算法具有平方加速效果：量子算法的复杂度为 $O(\sqrt{\beta/N})$，而经典算法的复杂度为 $O(\beta/N)$。

现在，我们可以将上述算法过程总结如下：

- 第一步：通过 H 门构造解空间，并得到解空间中所有基向量的平均叠加 $|\phi\rangle$
- 第二步：进行 Grover 迭代，对目标解向量的幅值进行放大
- 将第二步迭代 R 次，直至将 $|\phi\rangle$ 变换为足够接近目标 $|\phi_1\rangle$ 的 $G^R|\phi\rangle$

因此，现在只剩下了一个问题没有解决：在前面的分析中，我们必须预先知道满足要求的解向量个数 β，才能得到迭代次数 R，而 β 在实际情况中往往不可预知，这应当如何处理？我们将在下一节中介绍这个问题的一个解决方案。

II. θ 的估计

如前所述，在执行 Grover 迭代的时候，迭代次数 R 依赖于满足要求的解向量的个数 β，但在实际情况中，β 往往不能预先知道，这就造成了困难。事实上，一种基于相位估计的方法可以在指定误差内解决这个问题，本节我们将对此展开讨论。

前面提到，$\sin\theta = \sqrt{\beta/N}$，所以估计 β 可以转换为估计 θ 的值。如果我们求出 G 矩阵的特征值，会发现其特征值与对应的特征向量为：

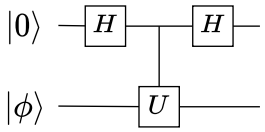
$$\lambda_\pm = e^{\pm 2i\theta}, \quad |\phi_\pm\rangle = \frac{|\phi_0\rangle \mp i|\phi_1\rangle}{\sqrt{2}}$$

这样，我们便可以使用相位估计的方法来以任意指定精度求解 θ。

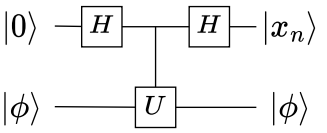
不失一般性，$0 \le \theta \le 2\pi$，现在假设我们用 n 位表示 θ，那么有：

$$\theta = 2\pi \cdot \sum_{j=1}^{N} \frac{x_j}{2^j}, \quad x_j \in \{0, 1\}$$

现在来介绍一种称为 Hadamard test 的结构，如下图所示：



如果我们令 $|\phi\rangle = |\phi_+\rangle$ 且 $U = G^{2^{n-1}}$，结合特征方程的性质 $G|\phi_+\rangle = \lambda_+|\phi_+\rangle$，可以得到：



这是因为

$$\begin{aligned} G^{2^n-1}|\phi_+\rangle &= e^{i\cdot(2^n-1)\theta}|\phi_+\rangle \\ &= e^{2\pi i\cdot(x_1x_2\cdots x_{n-1}.x_n)_2}|\phi_+\rangle \\ &= e^{2\pi i\cdot(0.x_n)_2}|\phi_+\rangle \\ &= \begin{cases} |\phi_+\rangle, & x_n = 0 \\ -|\phi_+\rangle, & x_n = 1 \end{cases} \end{aligned}$$

沿着这个思路，我们可以构造出如图 1 所示的量子线路，该量子线路可以以指定精度估计 θ，其中

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}$$

这样，我们就能得到指定精度下的 θ 的估计，据此可以得到目标解的个数 β。

III. 再看量子搜索

现在我们已经解决了量子搜索中出现的问题，理论上讲可以完整地进行 Grover 迭代并得到相应的结果。但是，使用基于相位估计的方法得到 β 的过程本身具有 $O(n^2)$ 的复杂度，这在实际使用时会消耗一定的时间和资源。

那么，我们很自然地就会提出这样一个问题：在目标元素占比 β/N 未知的情况下，能否设计出一种量子

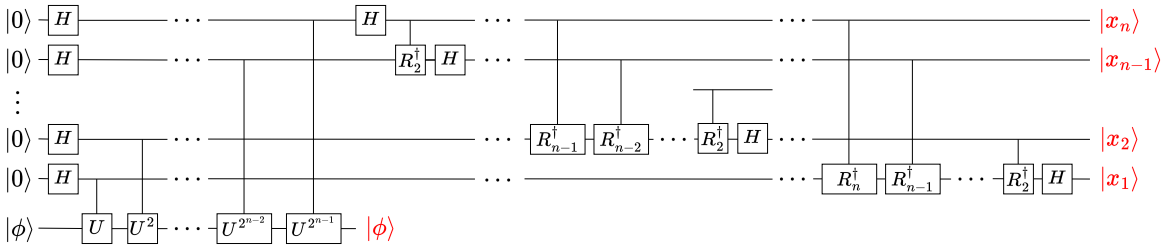


Figure 1. 估计 θ 的量子线路，精确到第 n 位

搜索算法，既可以找出所有的目标元素、又能保证平方加速?

如前所述，我们对 $|i\rangle$（$i = 0, 1, \cdots, 2^n - 1$）进行搜索，假设存在这样一种量子搜索算法，它在调用 t 次黑盒后可以精确地输出目标元素 i（如果无解则输出不合法的指标），那么该算法稍加修改就能以相同的黑盒调用次数计算 $F(x) = x_0 \vee x_1 \vee \cdots \vee x_{n-1}$（更一般地，我们可以计算 OR 之外的函数，比如 AND、PARITY 等），接下来，我们将黑盒写成下面的形式：

$$S_x|\rangle = (-1)^{x_i}|i\rangle = (1 - 2x_i)|i\rangle$$

容易发现：每次黑盒调用只会使任意基态前面的振幅多项式的次数增加 1，并且任意两次黑盒调用之间的酉变换作为线性变换并不会增加多项式的次数，所以在 t 次调用黑盒之后，任意基态前的振幅是阶不超过 t 的多项式，所以最终测得 1 的概率可以表示为次数不超过 $2t$ 的多项式。沿着这个思路，Beals 的一篇文章 [4] 证明了 $t \geq n/2$，这说明在目标元素占比未知的情况下，任意精确的量子搜索算法必定会丢失平方加速。

接下来我们来看在目标元素占比已知的时候有哪些常见的优化方法。

A. 修正最后一步

Brassard 等人提出了一种方法 [5]，这种方法直观上很好理解：既然原始 Grover 方法的误差来自于需要旋转的角度 $(\pi/2 - \theta)$ 不是单次旋转角度 2θ 的整数倍，那么在执行 $\lfloor k_{\text{opt}} \rfloor$ 次 Grover 迭代之后，将最后一步减小，便可以准确地到达目标位置。

具体来说，首先对初始状态向量 $|\phi\rangle = \mathcal{A}|0\rangle$ 作用 $\lfloor k_{\text{opt}} \rfloor$ 次 Grover 迭代，得到状态向量：$|\phi'\rangle = \cos((2\lfloor k_{\text{opt}} \rfloor + 1)\theta)|\phi_0\rangle + \sin((2\lfloor k_{\text{opt}} \rfloor + 1)\theta)|\phi_1\rangle$，然后考虑扩展 G 算子在 $\{|\phi_o\rangle, |\phi_1\rangle\}$ 基下的形式：

$$G = \begin{bmatrix} -(1 - e^{ix})\sin^2\theta - e^{ix} & e^{iy}(1 - e^{ix})\sin\theta\cos\theta \\ (1 - e^{ix})\sin\theta\cos\theta & e^{iy}((1 - e^{ix})\sin^2\theta - 1) \end{bmatrix}$$

再令 $G|\phi'\rangle$ 的 $|\phi_0\rangle$ 分量为 0，便能解出参数 x 和 y，使最终状态为 $|\phi_1\rangle$。

B. 借助三维球面

前一种方法是先按照标准的 Grover 迭代，最后一步做出相应的调整。那么另一个很容易想到的思路就是对每一步迭代都进行一些细微的调整，从而可以迭代得到目标状态。Long 在一篇文章 [6] 中提出了一种借助三维球面的方法，该方法将初始状态 $|\phi\rangle$ 视为 Bloch 球上的向量，而目标状态 $|\phi_1\rangle$ 在 Bloch 球上对应的向量为 $\overrightarrow{t} = (0, 0, 1)$，故我们的目的就是确定转轴 $\overrightarrow{l}$ 和每次的旋转角度 δ。

具体来说，首先将扩展 G 算子写成下面的形式：

$$e^{i\sigma}\left[\cos\left(\frac{\alpha}{2}\right)I + i\sin\left(\frac{\alpha}{2}\right)(n_x X + n_y Y + n_z Z)\right]$$

然后将扩展 G 算子展开为单位矩阵和泡利矩阵的线性组合，对比单位矩阵前的系数得到旋转角度 $\delta = \alpha = 4\beta$ $(\sin\beta = \sin(\sigma/2)\sin\theta)$；对比泡利矩阵前的系数得到转轴

$$\overrightarrow{l} = \frac{\cos\theta}{\cos\beta}(\cos(\sigma/2), \sin(\sigma/2), \cos(\sigma/2)\tan\theta)$$

可以证明，$\overrightarrow{s} - \overrightarrow{l}$ 和 $\overrightarrow{t} - \overrightarrow{l}$ 之间的角度为 $\omega = \pi - 2\beta$，而每一次旋转 $\delta = 4\beta$，因此有 $\omega = k\delta$，进一步可以得到：

$$\sin\left(\frac{\pi}{4k + 2}\right) = \sin\left(\frac{\sigma}{2}\right)\sin\theta$$

这样就得到了参数 σ 和旋转次数 k 之间的关系，通过选取合适的值，可以求解问题。

IV. 量子搜素的量子模拟角度解释

为了从更本质的角度来理解量子搜索问题，我们查阅了相关资料，发现有一种从量子系统演化角度来解释的思路 [8]，现简要介绍该思路。

假设我们有一个可用于搜索问题的哈密顿量 H，它依赖于解 x 和初始状态 $|\phi\rangle$，并在设定时间演化到目标状态 $|x\rangle$。如果我们构造哈密顿量 H 为

$$H = |x\rangle\langle x| + |\phi\rangle\langle\phi|$$

那么经过时间 t 后，初始状态为 $|\phi\rangle$ 的量子系统将演化到

$$\exp(-iHt)|\phi\rangle$$

对于很小的 t，演化的结果是将 $|\phi\rangle$ 变成 $(I - itH)|\phi\rangle = (1 - it)|\phi\rangle - it\langle x|\phi|x\rangle$，也就是说 $|\phi\rangle$ 稍微向 $|x\rangle$ 的方向发生了偏转。

接下来的问题就是确定演化时间 t。我们首先找到与 $|x\rangle$ 构成标准正交基的另一向量 $|y\rangle$，使 $|\phi\rangle = \alpha|x\rangle + \beta|y\rangle$，同时为了便于分析，取 $\alpha^2 + \beta^2 = 1$，所以在选定的基底 $\{|x\rangle, |y\rangle\}$ 下，有：

$$H = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} \alpha^2 & \alpha\beta \\ \alpha\beta & \beta^2 \end{bmatrix} = \begin{bmatrix} 1+\alpha^2 & \alpha\beta \\ \alpha\beta & 1-\alpha^2 \end{bmatrix}$$

忽略全局相位因子，可以得到 t 后系统的状态为

$$\cos(\alpha t)|\phi\rangle - i\sin(\alpha t)|x\rangle$$

所以，在 $t = \pi/2\alpha$ 时刻，对系统的观测以概率 1 得到结果 $|x\rangle$，也就是找到了问题的一个解。

现在的一个问题就是：时间 t 依赖于 α（即状态 $|\phi\rangle$ 在 $|x\rangle$ 上的分量），这是容易解决的，我们只需要取初始状态 $|\phi\rangle$ 为均匀叠加态，便对于任意的 $|x\rangle$ 都有 $\alpha = 1/\sqrt{N}$。同时，均匀叠加态可以用 Hadamard 门构造。

从量子模拟的角度来解释量子搜索算法，最大的意义就在于给出了 Grover 算法的一种“启发式”推导，这对于一些更复杂量子算法的研究是很有意义的。在利用量子模拟的方法得到了思路之后，我们只需要找到对应的量子电路来模拟演化所需的哈密顿量，这样便可以解决问题。我们可以将一般过程总结如下：（1）明确需要求解的问题，包括所期望的量子算法的输入输出描述；（2）猜测一个求解问题的哈密顿量，并证明其可行性；（3）找到一个过程来模拟这个哈密顿量。最后，如有需要，我们可以分析如时间复杂度等资源需求。

V. 量子搜索算法的应用

量子搜索算法有时对于求解 NP 问题具有显著的加速效果。对于某些特定的 NP 问题，我们有时可以设计出特定的量子搜索算法，从而实现经典计算机无法达到的加速效果。在后面的内容中，我们探讨了三个可以使用量子搜索算法进行加速的案例：图着色、整数分解和旅行商问题。

VI. 图着色问题

在图论中，图着色问题（Graph coloring problem）是图标注问题（Graph labeling problem）的特例，其描述如下：给定一个图形和一组 k 个标签（一般叫“颜色”），为图形的每一个顶点分配一个标签，以便不会有两个连接的顶点具有相同的标签。在图形上给顶点涂色是一种抽象的问题描述方式。实际上，这可以解决大量问题。例如，要分配的工作清单、用于在空间站之间进行通信的无线电频率的列表、或者提供给空间站的不同供应品的列表。

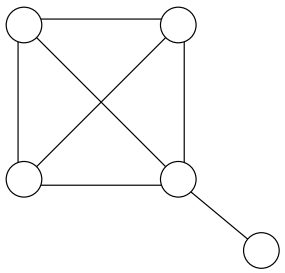


Figure 2. 图着色问题示例

例如，对图 2 中有 5 个顶点和 7 条边的图形，至少要用 4 种颜色才能标记。

图着色问题是一个 NP 完全问题，并且目前已知最好的对 n 个顶点进行 k 种着色的经典算法的运行次数有 $\Omega(2^n), k \geq 5$。采用 Grover 方法可以极大的加速该问题的解决。Grover 方法在上面已经进行详细的解决，在此不再重复。关于图着色问题部分的验证代码和结果详见附录 IX-1。下面将从颜色表示和经典算法开始，逐步介绍 Grover 算法对图着色问题的解法 [9]。

A. 颜色表示和操作

1) *初始化一个颜色的表示*：给定一个整数 $C(0 \leq C \leq 2^N - 1$，和一组 N 个 qubits，均为 $|0\ldots0\rangle$ 状态。将 qubits 数组制备为 C 的二进制表示基态。使用小端编码。例如，对 $N = 2$ 和 $C = 2$ 态应该为 $|01\rangle$。

首先将整数 C 转换到对应的二进制表示。在 Q# 中，可以用 “IntAsBoolArray” 函数将输入整数转化到对应小端二进制表示 “binaryC”。然后，我们需要使用 “binaryC” 作为位掩码 (bit mask)：无论何时 binaryC[i] 为 1，需要翻转对应的 qubit。这可以通过一个 X 门实现。

```
open Microsoft.Quantum.Convert;

operation InitializeColor (C : Int, register : Qubit[])
    : Unit is Adj {
    let N = Length(register);
    // Convert C to an array of bits in little endian
    format.
    let binaryC = IntAsBoolArray(C, N);
    // Value "true" corresponts to bit 1 and requires
    applying an X gate.
    for i in 0 .. N - 1 {
        if binaryC[i] {
            X(register[i]);
        }
    }
}
```

2) 从寄存器中读取颜色：输入为 N 个保证在 2^N 个基态中的 qunits，输出为 N-bit 整数，以小端方式代表这个基态。这个操作不会改变 qubit 的状态。例如，对 $N=2$，状态 $|01\rangle$ 返回 2（且 qubit 的状态不变，仍为 $|01\rangle$）。

由于确保了 register 在 2^N 个基态中，只需要简单的测量它，而不将 qubits 重置到 $|0\rangle$，这样可以保证 qubits 没有被改变，同时还给了我们必要的信息。可以使用 Q# 中的 MultiM 操作来测量每个 qubit，并将结果保存在数组中。

之后，将测量得到的 bits 转换到颜色对应的整数。

```
    operation MeasureColor(register : Qubit[]) : Int {
        let measurements = MultiM(register);
        return ResultArrayAsInt(measurements);
    }
```

3) 从寄存器中读取涂色方案：输入为在涂色中的元素数量 K，以及一个 $K\times N$ 的 qubit 数组，确保在 2^{KN} 个基态中。输出为一个 K N-bit 的整数数组，代表这个基态。数组中第 i 个整数被存储在下标为 $i\times N, i\times N+1,\ldots,i\times N+N-1$ 的 qubits 中，以小端方式存储。这个操作也不会更改 qubits 的状态。例如，对 $N=2, K=2$，qubit 状态 $|0110\rangle$，返回 [2, 1]。

这可以考虑为上一步骤的 K 次重复。在上一步骤中，我们将颜色从大小为 N 的寄存器中读取，这次我们需要对每个 N 位寄存器返回一个整数，共 K 个整数来代表 K 个颜色。

由于 K 已知，可以将寄存器的长度除以 K 来得到 N。然后，将 KN 个 qubit 的寄存器划分成 K 个 Nqubit 寄存器。最后，对每个分区进行上一步定义的 MeasureColor 操作，并将结果组成数组 coloring。

```
        operation MeasureColoring (K: Int, register :
    Qubit[]) : Int[] {
        let N = Length(register)/K;
        let colorPartitions = Chunks(N, register);
        let coloring = ForEach(MeasureColor,
    colorPartitions);
        return coloring;
    }
```

4) 2 位颜色等价黑箱：为了实现 K 种颜色对 N 个顶点图的着色，需要实现 N 位颜色等价黑箱。首先实现类似的 2 位颜色等价量子黑箱，输入为 2 个 qubits 组成的在任意状态 $|c_0\rangle$ 的数组，代表第一个颜色；以及 2 个 qubits 组成的在 $|c_1\rangle$ 的数组，代表第二个颜色；以及一个在状态 $|y\rangle$ 的 qubit（目标 qubit）。目标为将态 $|c_0\rangle|c_1\rangle|y\rangle$ 转换到态 $|c_0\rangle|c_1\rangle|y\oplus f(c_0,c_1)\rangle$（$\oplus$ 是模 2 加运算），其中，

$$f(x)=\begin{cases}1 & \text{if } c_0\text{和}c_1\text{在同一态下},\\ 0 & \text{otherwise.}\end{cases}$$

在操作结束后，寄存器保持初态。

由于已知 $f(c_0,c_1)=1$ 当且仅当 $c_0=c_1$，将其用 $\oplus$ 表示：

$$c_0=c_1 \iff c_0\oplus c_1=0$$

另一方面，如果 $c_0\oplus c_1=0$，可以表达 $f(c_0,c_1)$ 为 1。由于 $CNOT$ 门的存在，使用这种表示方式可以很方便的表示前一种形式。首先，分配一个额外的 $|0\rangle$qubit，将两个输入 qubit 分别作为控制位，额外 qubit 作为目标位，做两次 $CNOT$ 操作。由于 $CNOT$ 门的作用为 $|x\rangle|y\rangle\to|x\rangle|y\oplus x\rangle$，这样一对 $CNOT$ 控制门的效果是

$$|b_0b_1\rangle|0\rangle\to|b_0b_1\rangle|(0\oplus b_0)\oplus b_1\rangle$$

因此，通过分配两个 qubit$|a\rangle$ 到初始态 $|00\rangle$ 并对 c_0 和 c_1 的对应位使用 $\oplus$ 操作，我们可以计算出两个位串 c_0 和 c_1 的位 $\oplus$ 操作结果。

现在，只需当 $|a\rangle$ 为 $|00\rangle$ 态的时候翻转目标 qubit$|y\rangle$ 即可。通过使用由 $|0\rangle$ 控制的 X 门即可实

现。最后，需要取消计算按位异或以确保辅助量子位在释放他们之前再次处于 $|00\rangle$ 状态。

```
operation ColorEqualityOracle_2bit (c0: Qubit[], c1: Qubit[], target: Qubit): Unit is Adj+Ctl {
    use a = Qubit[2];
    within {
        // Compute bitwise XOR of c0 and c1,
        // Store result in a.
        CNOT(c0[0], a[0]);
        CNOT(c0[1], a[1]);
        CNOT(c1[0], a[0]);
        CNOT(c1[1], a[1]);
    } apply {
        // If all Xors are 0, c0=c1, and our function is 1.
        (ControlledOnInt(0, X))(a, target);
    }
}
```

5) N 位颜色等价量子黑箱：在上面对 2 位颜色等价量子黑箱的基础上，来实现 N 位颜色等价黑箱：对 N 位寄存器 c_0 和 c_1 进行位异或运算 ($\oplus$)，如果 XOR 为 0，则翻转目标量子位；首先，通过 CNOT 门，将 $c_1 \oplus c_1$ 存储在 c_1 中。将 c_0 的每个量子比特作为控制位，c_1 的对应量子位作为目标位，使用 N 个 CNOT 门实现上述过程，后面的过程与 2 位颜色等价量子黑箱一致。

```
operation ColorEqualityOracle_Nbit (c0: Qubit[], c1: Qubit[], target: Qubit):Unit is Adj+Ctl {
    within {
        // Compute bitwise XOR of c0 and c1 in place,
        // Store result in c1.
        for i in 0 .. Length(c0) - 1 {
            CNOT(c0[i], c1[i]);
        }
    } apply {
        // If all XORs are 0, c0 = c1, and our function is 1.
        (ControlledOnInt(0, X))(c1, target);
    }
}
```

上述过程完成后，图着色问题的基本描述工具就准备完成了。

B. 图着色

1) 图着色问题的经典验证方式：给定图中顶点数量 $V(V \leq 6)$，一个含有 E 个元组的整数数组，代表图中边的数量 ($E \leq 12$)，每个元组给定边的两个顶点下标，下标范围为 0 到 $V-1$，还有一个 V 个整数的数组，代表图的顶点着色。第 i 个元素代表第 i 个顶点的颜色。判断所给的图着色是否正确（即没有一条边的两个顶点同色）。

要验证一种着色方法是否合理，需要检查每一条边，判断是否两个顶点的着色相同。如果相同，则着色不合理。如果每条边都通过测试，则说明着色方式有效。由于顶点 n 的颜色是 colors 数组的第 n 个元素，只需要遍历该数组，并比较对应颜色即可。

```
function IsVertexColoringValid(V: Int, edges: (Int, Int)[], colors: Int[]):Bool {
    // Loop through every edge.
    for(v0, v1) in edges{
        // Compare the colors of vertives.
        if colors[v0] == colors[v1] {
            return false;
        }
    }
    // If all edges are valid, then this vertex coloring is valid.
    return true;
}
```

2) 验证着色方案的量子黑箱：在上面的基础验证着色方案的方法的基础上，将其修改为量子黑箱方法。除了给定顶点数量 V ($V \leq 6$)，以及 E 个元组的整数数组代表图的 E 条边 ($E \leq 12$) 之外，还给定 $2V$ 个 qubits 数组，编码颜色安排方案，以及一个任意状态的 qubit$|y\rangle$ 作为目量子位。

目标为将态 $|x, y\rangle$ 转换到态 $|x, y \oplus f(x)\rangle$ ($\oplus$ 为模 2 加)。当所给的顶点着色方案合理的时候，$f(x) = 1$；否则为 0。保持查询寄存器的状态不变。

在上面节 VI-A5 中实现的 “ColorEqualityOracle_Nbit” 操作的基础上，我们将申请一个量子比特数组，与边数组的大小一致。如果对应边连接的顶点有同一种颜色，则翻转对应量子比特。在比对结束后，如果数组仍在状态 $|0 \ldots 0\rangle$，则说明这种涂色方案是合理的。

由于涂色方案由量子比特数组的方式给出，每个顶点 2 个量子比特 (2 量子比特 =4 基态 =4 颜色)，我们需要采用正确的着色块。可以推断出顶点 n 的着色是在 $2 \times n$ 和 $2 \times n + 1$ 的量子比特中编码的。在计算结束后，对量子比特解除计算。

3) 实现图形着色问题的量子黑盒：有一张由边连接的点的图，考虑如何对其进行着色。为此，首先实现量子黑盒。

```
    operation VertexColoringOracle (V: Int, edges: (Int, Int)[], colorsRegister: Qubit[], target: Qubit): Unit is Adj+Ctl{
        // Store the number of edges.
        let edgesNumber = Length(edges);
        // Allocate the array of qubits storing the conflicts of the coloring.
        use conflicts = Qubit[edgesNumber];
        within {
            // Iterate over every edge.
            for i in 0 .. edgesNumber - 1{
                // Deconstruct the edge tuple into two separate vertex indices.
                let (v0, v1) = edges[i];
                // Track conflicts.
                ColorEqualityOracle_Nbit(colorsRegister[2*v0 .. 2*v0+1], colorsRegister[2*v1 .. 2*v1+1], conflicts[i]);
            }
        }
        apply {
            // If all the edges are valid, conflicts should be in state |0...0>
            (ControlledOnInt(0, X))(conflicts, target);
        }
    }
```

解决了判断着色方案正确性的问题后，可以使用 Grover 搜索方法寻找顶点着色方案。

4) Grover 搜索方法寻找顶点着色方案：给定图中顶点数量 V ($V \leq 6$)，以及上面实现的顶点着色验证方法，找寻一种合法的图顶点着色方案。

为了实现图着色算法，需要实现一般的 Grover 算法。首先对上面的顶点着色验证量子黑盒进行封装，将其变成阶段量子黑盒。阶段量子黑盒可以通过根据 O 的输入应用相位，将 f 编码成预言 O。然后实现 Grover 算法。

Listing 1. 将标记量子黑盒转为阶段量子黑盒

```
    operation OracleConverter (markingOracle: ((Qubit[], Qubit)=>Unit is Adj), register:Qubit[]) : Unit is Adj {
        use target = Qubit();
        within {
            // Put the target qubit in the |-> state.
            X(target);
            H(target);
        } apply {
            // Apply the making oracle
            markingOracle(register, target);
        }
    }
```

Listing 2. Grover 算法

```

    operation GroverAlgorithmLoop (markingOracle: ((Qubit[], Qubit)=>Unit is Adj), register: Qubit[], iterations: Int): Unit is Adj {
        // Convert the marking oracle in a phase oracle
        let phaseOracle = OracleConverter(markingOracle, _);
        // Prepare an equal superposition of all basis states
        ApplyToEachA(H, register);
        // Apply Grover iterations.
        for _ in 1..iterations {
            // Apply phase oracle.
            phaseOracle(register);
            // Apply "reflection about the mean".
            within {
                ApplyToEachA(H, register);
                ApplyToEachA(X, register);
            } apply {
                (Controlled Z)(Most(register), Tail(register));
            }
        }
    }
```

要使用此 Grover 算法找到图着色问题的解，需要在通用实现的基础上做额外的步骤：

- 分配一个量子比特数组来存储图着色方案，每个顶点 2 个量子比特和一个额外的量子比特来用来验证我们找到的图着色分配方案是否正确
- 尝试按不同的 Grover 迭代次数运行此算法，从 1 次迭代开始，每次找不到一个解的时候增加迭代次数。我们将使用两个变量 “iterations” 和 “correct”，分别存储迭代次数和表示是否找到了正确的解
- 在循环体中，进行如下步骤：
  - 执行目前数量的迭代次数的 Grover 算法循环
  - 测量 Grover 算法得到的量子比特数组，可以使用 Q# 中的 MultiM 操作
  - 通过对存储涂色方案和额外量子比特的量子比特数组应用标记量子黑盒操作来判断是否结果是合理的涂色方案。测量结束后，这些量子比特的状态会坍缩到对应测量结果的基态

  - 测量在 Pauli Z 基态下的额外量子比特位，并将其使用 “MResetZ” 操作重置
    * 如果测量结果为 “One”，则算法结果是正确的方案，需要将“correct”变量设为“true”，并将结果使用前面实现的 “MeasureColoring” 操作解码为一种图着色方案
    * 如果测量结果为 “Zero”，则算法结果不是正确的方案，不做任何操作
  - 重置存储涂色方案的数组，使其准备好下一次迭代
- 如果我们找到一种方案，或者运行的迭代次数太多，我们停止循环（在运行迭代次数太多的情况下，即没有找到合理的涂色方案）

据此实现的 Grover 图着色算法代码如下。

```
    operation GroversAlgorithm(V: Int, oracle: ((Qubit[], Qubit)=>Unit is Adj)):Int[] {
        mutable coloring = [0, size=V];
        use (register, output) = (Qubit[2*V], Qubit());
        mutable correct = false;
        mutable iterations = 1;
        repeat {
            Message($"Trying iteration {iterations}");
            GroverAlgorithmLoop(oracle, register, iterations);
            let temp = MultiM(register);
            oracle(register, output);
            if MResetZ(output) == One {
                set correct = true;
                set coloring = MeasureColoring(V, register);
            }
            ResetAll(register);
        }
        until(correct or iterations > 10)
        fixup {
            set iterations += 1;
        }
        if not correct {
            fail "No valid coloring was found";
        }

        return coloring;
    }
```

VII. 整数分解问题

A. 问题介绍

整数分解（integer factorization）又称素因数分解（prime factorization），该问题是将一个正整数分解成几个素数的乘积形式。整数分解问题 (IFP) 的描述如下：

$$\text{IFP} = \begin{cases} \text{输入} & n \in \text{合数} \\ \text{输出} & a \quad \text{使} a|n \text{且} 1 < a < n \end{cases}$$

整数分解问题已有数百年的历史，是数论中的经典问题之一。最近几十年，随着信息技术的不断发展，整数分解问题已经延伸到通信、密码学等多个领域。给出两个约数可以很容易得到它们的乘积，但是根据其乘积分解出相应的因子则并不容易。这便是很多现代密码系统的关键所在，能够快速求解出整数的分解问题便能攻破这些密码系统。因此，整数分解问题的快速求解算法也深受密码学界、计算机学界的广泛关注。

整数分解问题既未被证明是多项式时间可解的 P 问题，也未被证明是 NP 完备问题。对于该问题，现如今主要有经典算法和量子算法两种方式，由于量子技术的叠加性和相干性，使得量子并行处理可以极大地提高计算的效率，从而完成经典计算机无法完成的工作。Shor 算法和 Grover 搜索算法是目前对于解决整数分解问题最高效的量子算法。

B. 经典算法

下列算法针对整数 n，取得大素数 p 和 q，使得 $n = pq$。

1) *试除法*：该方法是整数分解最简单最直接的方法。对于整数 n，首先从最小素数 2 开始试除，若能够整除，则依次再用其他素数重复试除过程，直到最后的商是素数，这便是试除法求解整数分解的思路。

假设分解整数 n 用到 2 到 $\sqrt{n}$ 的所有素数来试除，则只需尝试 $O(\sqrt{n}/\log n)$ 个素数，便可求得整数 n 的素因子，即试除法的时间复杂度最快为 $O(\sqrt{n}/\log n)$。

但试除法只适用于分解整数和素因子较小的情况，对于分解的整数较大或者素因子较大时，试除法的效率较为低下。

2) *椭圆曲线分解法*：椭圆曲线分解法 [10] 是荷兰科学家 Lenstra H. 于 1987 年提出的整数分解法。根据椭圆曲线的群结构，随机选取一椭圆曲线 $E: y^2z = x^3 + axz^2 + bz^3$，若 (x, y, z) 在该椭圆上，且 $c \neq 0 \bmod p$，则 (cx, cy, cz) 也在该椭圆上，即满足该方程，因此 (x, y, z) 和 (cx, cy, cz) 可看成等价的。用 $(x : y : z)$ 表示包含 (x, y, z) 一类等价点的等价类。在群 $E_(a, b)$ 中的加法零元 O 是 $(1 : 0 : 1)$，此时 $z \equiv 0 \bmod p$，

若在某一步运算中得到了加法零元 $O=(x,y,z)$，那么计算 $\gcd(z,n)$ 所得的因子，可将整数 n 分解。椭圆曲线分解法的时间复杂度为 $\exp((1+o(1))\sqrt{\log n \log\log n})$，其中 p 为 n 的最小素因子。

3) 费马分解法：费马分解法适合分解含有几乎相等的两个因子的一类整数 n。若 n 是两个整数的平方差，即 $n=a^2-b^2$，则 $n=(a+b)(a-b)$ 是整数 n 的一个分解。将 n 表示成平方差，可令 $b=1,2,3\ldots$，然后看 $n+b^2$ 是否为完全平方数。如果 $n+b^2$ 为完全平方数，那么便得到了 n 的一个分解。

4) 数域筛选法（GNFS）：数域筛选法 [11] 由 Pollard J. 提出，是目前的经典算法中最快的整数分解算法。数域筛选法包括多项式的生成、数域的生成、分解基的生成、筛过程、分解基矩阵的生成、同态映射、求根等步骤。该算法以解 $a^2\equiv b^2 \pmod n$ 为目的，令 $f(x)$ 为一正系数多项式，m 为一有理数，有 $f(x)\equiv 0 \pmod n$，α 为多项式 $f(x)$ 的一复根，定义一个从 $R=Z[\alpha]$ 到整数域的一个环同态 $\varphi: R\to Z_n \varphi(\alpha)=m \pmod n$，则有：

$$x^2=\varphi(\beta)^2=\varphi(\beta^2)=\varphi(\prod_i a_i+b_i\alpha)$$
$$=\prod_i (a_i+b_i m)=Y^2 \pmod n$$

所得同余式 X，Y 即为所需。要得到满足 $\prod_i(a_i+b_i\alpha)=\beta^2$ 和 $\prod_i(a_i+b_i m)=Y^2$ 的整数对 (a,b)，需将 $Z[\alpha]$ 上的平滑和 Z 上的平滑概念结合，整数对 (a,b) 满足 $a+b\alpha$ 在 $Z[\alpha]$ 的代数因子库上平滑，同时 $a+bm$ 在 Z 的有理因子库上平滑。当找到足够多的在两因子库上都平滑的整数对后，在 $Z[\alpha]$ 和 Z 上同时产生一个平方值。数域筛选法的时间复杂度为 $O(\exp[(\log\ N)^{1/3}(\log\ (\log\ N))^{2/3}])$。

C. 量子算法

1) 基于 Grover 搜索算法的整数分解：Grover 算法 [12] 就是要通过寻找一个酉变换来反复迭代增大要找的量子态 $|\psi\rangle$ 的概率，同时减少 $|x\neq\psi\rangle$ 态的概率。首先将整数分解问题转化为二进制形式：用二进制表示整数 N、因子 P 和 Q，根据 N、P 和 Q 之间的关系通过二进制乘法建立方程，则整数分解问题即转化为求解因子 P 二进制的第 i 位 p_i 和因子 Q 二进制的第 j 位 q_j。

Grover 算法求整数分解问题步骤可以概括如下：

- 根据二进制相乘得到方程组并化简；

$$p_1+q_1=1$$
$$p_2+q_2=1$$
$$p_2q_1+p_1q_2=1$$

此时，变量 p_1 、p_2、q_1 、q_2 都为二元域。

- 初始化制备等权叠加态，通过 Hadmard 门将量子基态转化为等权叠加态：

$$|s\rangle=H^{\oplus n}|00\cdots0\rangle=1/\sqrt{2^n}\sum_{x=0}^{2^n+1}|x\rangle$$

- 将步骤（1）中的方程转化为布尔关系式：

$$(p_1\vee q_1)\wedge(p_2\vee q_2)\wedge[(p_1\vee q_2)\vee(p_2\vee q_1)]$$
$$=(p_1\wedge\neg p_2\wedge\neg q_1\wedge q_2)\wedge(\neg p_1\wedge p_2\wedge q_1\wedge\neg q_2)$$

再根据这些逻辑关系构造 $Oracle(U_w)$，其中包括两个目标项。

- 构造反演算子 (U_s)，增加通过该算法搜索到目标项的概率。反演算子的公式为：

$$U_s=2|s\rangle\langle s|-I$$

- 重复迭代。$Oracle(U_w)$ 和反演算子 (U_s) 共同构成 Grover 算法的 G 迭代，通过重复 G 迭代可以改变搜索到目标项的概率。当 G 迭代的次数接近 $\lceil\frac{\pi}{4}\sqrt{\frac{N}{M}}\rceil$ 时，可以认为 Grover 算法搜索到目标项的概率为最优，其中 N 为待搜索元素总个数，M 为目标项元素个数。

下面进行 Grover 算法效果分析。

Grover 搜索算法是一种通用的量子搜索算法，其时间复杂度为 $O(\sqrt{n})$，为整数分解等搜索问题的解决提供了平方根加速，相当于将搜索路径减半，从而实现了高效提速，并节约了大量的计算资源。

随着 Grover 算法迭代次数的增加，线路所需的量子比特数不变，但线路深度越来越深。如何优化算法的线路深度，使其在资源受限情况下高效执行，将是该算法未来发展的关键问题之一。

2) Shor 算法简介及对比：Shor 于 1994 年第一个提出了量子整数分解算法 [13]，掀起了量子计算机研究的高潮。整数分解问题在经典计算机上需要指数时间才

能完成，而 Shor 算法在量子计算的条件下，使得整数分解问题可以在多项式时间内便可得到解决。

Shor 量子整数分解算法的核心是利用一个隐藏在因子分解问题中的特殊结构，该结构允许把整数分解问题转化为求一个特定函数周期的问题 [14]。该算法的关键思想在于将分解问题转化为模指数线路的周期问题，构建模指数线路，通过逆量子傅里叶变换（QFT）找到模指数线路的周期。

接下来我们对比基于 Grover 算法和 Shor 算法的整数分解方法。

Shor 算法将整数分解转化为特定函数周期问题，对于整数分解可以起到指数级加速的效果，将问题规模降至 $O(\log N)$；而 Grover 算法则将问题转化为 N 元素无结构数据库搜索问题，该算法主要依赖于核心模块的多次迭代，解决问题的规模为 $O(\sqrt{n})$。此外，两种算法对于分解不同规模数字所用的量子比特数如下表 I 所示 [15]：

Table I
两种算法分解不同规模数字所用量子比特数

N	Shor 算法	Grover 算法
	量子比特数	
143	18	9
3599	26	9
25217	32	9
110633	36	9
731021	42	9
1520273	44	9
16850989	52	9

Shor 算法所用量子比特数随整数的变大增速明显，而 Grover 算法在处理相同规模数字时所用的量子比特数目一般较少，所以 Grover 对量子计算机的水平要求更低，更为通用，能够节约计算资源，更加切实可行。

D. 意义和应用

整数分解问题在代数学、密码学、计算复杂性理论等领域中有着重要的意义。目前世界上公认的最成熟完善的公钥密码体制 RSA 算法 [16] 便是基于整数分解问题而设计的，RSA 能够抵挡绝大多数的密码攻击。而随着 Shor 算法的提出，RSA 的安全性便受到了威胁，但由于目前的量子计算机尚在发展过程中，且 Shor 算法的随机性大，效率不高，因此 Shor 算法也不能有效地利用于破解 RSA。如今，随着对量子计算机水平要求更低的 Grover 量子整数分解算法的不断发展和改进，Grover 搜索算法对于搜索问题的平方根加速相当于将密钥长度减半，在理论上将有机会破解目前被公认为最安全的 RSA 公钥密码。

公钥密码体制在量子计算环境下的安全性分析无论是在理论上还是实际应用中都具有重大研究意义，特别是广泛使用的 RSA、ElGamal 和 ECC 等公钥密码体制的安全性更值得我们深入研究。

VIII. 旅行商问题

A. 问题介绍

旅行商问题（TSP）描述了以下情景：给定一个含有 n 个城市的城市列表，并给定每个城市与其余所有城市相连接的道路长度矩阵，如何为旅行推销员选取一条最短的可能路线，使得推销员正好访问每个城市一次并返回起点城市。

TSP 是经典的 NP-hard 问题中的一个。随着问题中元素数量的增加，即旅行商问题中城市数量的增加，导致找到问题的解决方案所需的时间呈指数增长，也就意味着 TSP 问题在经典计算机上难以解决。对于一个 n 足够大的旅行商问题，经典计算机和应用于经典计算机上的算法无法解决这一问题。

为了检验量子搜索算法对求解 NP 问题的加速效果，本部分给出关于如何采用量子搜索算法解决一类典型 NP-complete 问题，并验证其针对经典算法的加速效果和算法有效性。讨论关于采用量子搜索算法解决这一问题的可能现实应用与前景展望。

B. 背景知识

1) NP-complete 问题: 决策问题包含 P 问题和指数时间问题，P 问题为一类可以通过确定性图灵机在多项式时间内解决的问题集合，而指数时间问题在经典计算机上随着元素数量增加，解决问题时间呈指数增长。

NP 问题是指数时间问题的子类问题，为一类可以通过非确定性图灵机在多项式时间内解决的问题集合。

NP-complete 问题则是 NP 问题的子类问题，并且所有的 NP 问题最终都可以规约成 NP-complete 问题，这也就意味着如果能在多项式时间内解决 NP-complete 问题，则所有 NP 问题也可在多项式时间解决。

NP-complete 问题包罗甚多，但 3-satisfiability、3-dimensional、vertex cover、clique、partition、Hamiltonian circuit 等六个问题构成了 NP-complete 问题的代

表性集合。而 Hamiltonian circuit 即为本部分所讨论的旅行商问题的核心，两者在解决思路上基本一致。

2) TSP 问题整数规划模型: 首先介绍 Dantzig-Fulkerson-Johnson（DFJ）模型。

假定旅行商问题中 n 个城市的城市列表为 V, 城市间的距离矩阵为 A，建立以下基于平面分割法的 DFJ 模型。定义决策变量

$$x_{ij} = \begin{cases} 1, & \text{如果有边从}i\text{到}j \\ 0, & \text{否则} \end{cases} \qquad (i,j) \in A$$

模型目标函数可以写作

$$\min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}$$

并满足下述约束条件

$$\begin{aligned} \sum_{i \in V,(i,j) \in A} x_{ij} &= 1 \quad \forall j \in V \\ \sum_{j \in V,(i,j) \in A} x_{ij} &= 1 \quad \forall i \in V \\ \sum_{i \in S} \sum_{j \in S} x_{ij} &\leq |S| - 1 \quad \forall S \subset V, S \neq \emptyset \end{aligned}$$

但 DFJ 模型约束规模过大，无法求解大规模的算。接下来我们介绍 Miller-Tucker-Zemlin（MTZ）模型。

MTZ 模型相比于 DFJ 模型增加一组变量以确定点的顺序来消除子回路。MTZ 模型定义决策变量如下:

$$\begin{cases} 1, & \text{如果有边从}i\text{到}j \\ 0, & \text{否则} \end{cases} \qquad (i,j) \in A$$

模型目标函数可以写作:

$$\min \sum i \in V \sum_{j \in V} c_{ij} x_{ij}$$

并满足下述约束条件

$$\begin{aligned} \sum_{i \in V,(i,j) \in A} x_{ij} &= 1 \quad \forall j \in V \\ \sum_{j \in V,(i,j) \in A} x_{ij} &= 1 \quad \forall i \in V \\ u_j \geq u_i + n(x_{ij} - 1) + 1 &\quad \forall i,j \in V - \{0\} \\ x_{ij} \in \{0,1\} &\quad \forall i,j \in V \\ u_i \in \mathcal{R}_+^0 &\quad \forall i \in V \end{aligned}$$

C. 经典算法及其时间复杂度

1) 蛮力法: 蛮力法，顾名思义也可以视作为枚举法，通过找到所有顶点的每个可能组合来解决问题。本算法的时间复杂度为

$$O(n!), \quad \text{其中}n\text{为顶点数}$$

2) 动态规划法: 对于图 $G = (V, E)$，假设从顶点 i 出发，令 $V' = V - i$，则用 $d(i, V')$ 表示从顶点 i 出发经过集合 V' 中各个顶点各一次且仅一次，最后回到出发点 i 的最短路径长度。所以子问题: 用 $d(k, S$ 表示从顶点 k 出发经过集合 S 中各个顶点各一次且仅一次，最后回到出发点 i 的最短路径长度。

所以 TSP 问题的状态转移方程可以表示为:

$$\begin{aligned} d(i, V') &= \min_{k \in V'} \{c_{ik} + d(k, V' - \{k\})\} \quad (k \in V') \\ d(k, \{\}) &= c_{ik} (k \neq i) \quad (c_{ik} \in E) \end{aligned}$$

该算法的时间复杂度为

$$O(n^2 2^n), \quad \text{其中}n\text{为顶点数}$$

3) 分支限界法: 分支限界法通过应用剪枝函数广度遍历解空间树，直到产生一个接近最优值的解，该算法的事件复杂度为

$$O(n^2 2^n), \quad \text{其中}n\text{为顶点数}$$

4) 近似最优值求解: 如上所述，经典计算机对于 NP 问题无法在多项式时间内给出精确解，但可以通过近似的方法在牺牲准确性的前提下提升计算速度，其中一种近似，可以在多项式时间内给出小于等于 1.5 倍最短路径长度的结果。

该算法主要思想为在多项式时间内找到最小生成树，并添加一个最小权重匹配，使其成为欧拉图。最终的欧拉图长度为不超过最小旅行商行程长度 1.5 倍的值。

D. 基于量子搜索的相位估计算法

1) 算法整体思想: 我们可以将城市间距离抽象为相位，并根据给定的城市间距离矩阵构建一个特征值是上述相位组合的单元算子。将相位估计算法应用于某些特征态，得出这些特征态所表示的汉密尔顿回路的长度值。最后利用现有的量子搜索算法从得到的长度值中找到最小值并得出最小距离相对应的路线规划。

2) 算法实现: 具体而言，对于经典算法中输入的矩阵构建其相应的相位矩阵 $[A]_{ij}=\phi_{ij}$，其中 ϕ_{ij} 表示从城市 i 到城市 j 的代价。但由于直接构建的相位矩阵 $[A]_{ij}$ 往往不是酉矩阵，所以无法在量子计算机上实现和操作该运算符；除此之外，当矩阵相乘或以上述相位作为系数的态做张量积时，相位会被添加。所以可以将输入的矩阵进一步表示为矩阵 B，其中 $B_{ij}=e^{i(\phi_{ij})}$，这样矩阵 B 的对角线元素全为 1，即为单位算子。

为了后续的相位估计，由矩阵 B 为每一个城市（节点）构造酉矩阵 U_j

$$[U_j]_{kk}=\frac{1}{\sqrt{n}}[B]_{jk}$$

其中 n 为城市数量，$j,k\in[1,n]$，矩阵 U_j 剩余元素均初始化为 0，由此可知 U_j 为对角矩阵。例如 U_1 为（假设城市数量为 4）

$$e^{i\phi_{11}}|00\rangle\langle 00|+e^{i\phi_{21}}|01\rangle\langle 01|$$
$$+e^{i\phi_{31}}|10\rangle\langle 10|+e^{i\phi_{41}}|11\rangle\langle 11|$$

相位估计算法通过相位反冲将酉矩阵的相位写入 t 个量子比特中，当通过一个量子比特控制 U 门时，由于相位反冲量子比特将与相位成比例旋转 $e^{2i\pi\theta}$。所以需要通过受控酉 C-U 使得只有在目标寄存器的控制位为 $|1\rangle$ 时才进行 U 操作，进而需要将 U_j 分解为受控酉 CU_j，也就意味着要得到整体酉 U

$$U=U_1\otimes U_2\otimes\cdots\otimes U_n$$

进而得到

$$CU=C(U_1\otimes U_2\otimes\cdots\otimes U_N)=CU_1\otimes CU_2\otimes\cdots\otimes CU_n$$

对于特征态，由于 U 为对角矩阵，所以其中的 n^n 个元素只有 $(n-1)$ 个元素包含 TSP 中所有不同的汉密尔顿回路长度的可能值，而每个特征向量表示不同的汉密尔顿回路。所以 U 的 $(n-1)!$ 个可能的特征态的特征值是相应的汉密尔顿回路的总成本。对于按照某种顺序通过所有城市并抵达其实状态的路线的相位，用下列函数将城市转化为计算基矢量（其中特征态以二进制形式表示）:

$$|\psi\rangle=\otimes_j|i(j)-1\rangle$$

其中 $j\in[1,n]$，函数 $i(j)$ 表示从哪座城市到达城市 j。

通过上述过程可以得到 $(n-1)!$ 个特征向量，通过

$$L=U|\psi\rangle$$

可以计算出所有可能的 $(n-1)!$ 条汉密尔顿回路的程度 L。最后，只需要通过已有的量子搜索算法，例如 Grover 算法，在已得到的 $(n-1)!$ 条汉密尔顿回路中找到最小值以及所对应的特征向量，即可得到旅行商问题的最小周游路径长度和路线规划。

根据 Grover 量子搜索算法可知，本算法的时间复杂度为

$$O(\sqrt{(n-1)!},\quad \text{其中}n\text{为城市数量 (节点数量)}$$

3) 算法有效性分析: 相位估计算法在得到所有可能的汉密尔顿回路后，采用量子搜索算法（Grover 算法）从中找出最小值和最优解，这一搜索部分的时间复杂度为 $O(\sqrt{(n-1)!})$，相比于经典的蛮力求解具有二次方加速的效果，在规模较小时本算法优于经典算法。

但是相位估计算法仍然没有提供一个完美的解决方案，因为并未给出有效的算法来在多项式时间内解决这一 NP-hard 问题。除此之外，在 TSP 求解的过程中找到所有可能的汉密尔顿回路自身就是一个 NP-complete 问题，随着规模扩大，寻找所有可能的汉密尔顿回路就是一项繁琐的工程。

E. 其他技术路线的量子算法

针对一个每个顶点的度均为 k 的图，Mandra 等人通过将问题化简为占位问题，通过量子回溯算法对回溯过程进行加速，在 $O(2^{(k-2)n/4})$ 的时间内给出汉密尔顿回路 [20]。当 $k=3$ 时，该算法的上界为 $O(1.189^n)$；当 $k=4$ 时，该算法的上界为 $O(1.414^n)$，当 $k\geq 5$ 时，本算法就不再优于某些经典算法。

除此之外，Goswami 等人给出了一个有效解决近似旅行商问题的框架 [21]。Bang 等人通过泛化 Grover 搜索给出了解决 TSP 的一种量子启发式算法 [22]。但这些算法均有一定的限制条件，同时也无法真正做到在多项式时间给出旅行商问题的解。

F. 现实意义与应用

1) 实际应用: 具有大搜索空间的高维优化问题是许多现有科学问题或行业瓶颈的关键因素之一。对于商业中的优化问题，旅行商问题的解决可以有效改善物

流选择，在动态条件影响采购和物流选择时，受限优化通过确定最佳的前进路线来帮助供应链的生成。同时，TSP 的解决也将为各类目前难以解决的优化问题提供思路和解决基础。

2) 理论计算机科学与运筹学: TSP 为典型的 NP-hard 问题，而作为 TSP 核心的汉密尔顿回路问题则是 NP-complete 问题的六大代表性问题之一。解决 TSP 问题的意义远不止具体问题的解决，随着 TSP 问题在多项式时间内解决的算法被给出，由其衍生的旅行者问题、车辆路线问题将迎刃而解，也将使所有 NP 问题在多项式时间内可解。而随着 NP 问题的解决也会导致计算机界诸多既定现实的改变，也会进一步推动数学、优化、人工智能、生物学、物理学、经济学和工业领域等的发展，为发展提供更宽广的可能性。

References

[1] Grover, Lov K. "Quantum mechanics helps in searching for a needle in a haystack." Physical review letters 79.2 (1997): 325.

[2] Deutsch, David, and Richard Jozsa. "Rapid solution of problems by quantum computation." Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences 439.1907 (1992): 553-558.

[3] Grover, Lov K. "Fixed-point quantum search." Physical Review Letters 95.15 (2005): 150501.

[4] Beals, Robert, et al. "Quantum lower bounds by polynomials." Journal of the ACM (JACM) 48.4 (2001): 778-797.

[5] Brassard, Gilles, et al. "Quantum amplitude amplification and estimation." Contemporary Mathematics 305 (2002): 53-74.

[6] Long, Gui-Lu. "Grover algorithm with zero theoretical failure rate." Physical Review A 64.2 (2001): 022307.

[7] 李冠中, 李绿周. " 精确 Grover 量子搜索算法概述." 电子科技大学学报 (2022).

[8] Nielsen, Michael A., and Isaac Chuang. "Quantum computation and quantum information." (2002): 558-559.

[9] A. Saha, D. Saha and A. Chakrabarti, "Circuit Design for K-coloring Problem and its Implementation on Near-term Quantum Devices," in 2020 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS), Chennai, India, 2020 pp. 17-22.

[10] Lenstra Jr, Hendrik W. "Factoring integers with elliptic curves." Annals of mathematics (1987): 649-673.

[11] Lenstra, Arjen K., et al. "The number field sieve." The development of the number field sieve. Springer, Berlin, Heidelberg, 1993. 11-42.

[12] 刘晓楠, et al. "Grover 算法改进与应用综述." 计算机科学 48.10 (2021): 315-323.

[13] Shor, Peter W. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." SIAM review 41.2 (1999): 303-332.

[14] Hardy, Godfrey Harold, and Edward Maitland Wright. An introduction to the theory of numbers. Oxford university press, 1979.

[15] 宋慧超, et al. " 基于 Grover 搜索算法的整数分解." 计算机科学 48.4 (2021): 20-25.

[16] 涂玲英, et al. " 对 Shor 算法破解 RSA 的探讨." 华侨大学学报: 自然科学版 36.6 (2015): 640-644.

[17] Srinivasan, Karthik, et al. "Efficient quantum algorithm for solving travelling salesman problem: An IBM quantum experience." arXiv preprint arXiv:1805.10928 (2018).

[18] Hopper, Buckley. "Searching for a Quantum Algorithm to Solve the Traveling Salesman Problem." (2001).

[19] Jain, Siddharth. "Solving the traveling salesman problem on the d-wave quantum computer." Frontiers in Physics (2021): 646.

[20] Mandra, Salvatore, Gian Giacomo Guerreschi, and Alán Aspuru-Guzik. "Faster than classical quantum algorithm for dense formulas of exact satisfiability and occupation problems." New Journal of Physics 18.7 (2016): 073003.

[21] Goswami, Debabrata, et al. "Towards efficiently solving quantum traveling salesman problem." arXiv preprint quant-ph/0411013 (2004).

[22] Bang, Jeongho, et al. "A quantum heuristic algorithm for the traveling salesman problem." Journal of the Korean Physical Society 61.12 (2012): 1944-1949.

[23] Moylett, Alexandra E., Noah Linden, and Ashley Montanaro. "Quantum speedup of the traveling-salesman problem for bounded-degree graphs." Physical Review A 95.3 (2017): 032323.

IX. Appendix A

Listing 3. 图着色问题测试

1) 图着色问题测试代码和测试结果:

```
// Hardcoded graphs used for testing the vertex coloring problem:
//  - trivial graph with zero edges
//  - complete graph with 4 vertices (4-colorable)
//  - disconnected graph
//  - random connected graph with more edges and vertices (3-colorable)
//  - regular-ish graph with 5 vertices (3-colorable, as shown at https://en.wikipedia.org/wiki/File:3-coloringEx.svg without one vertex)
//  - 6-vertex graph from https://en.wikipedia.org/wiki/File:3-coloringEx.svg
function ExampleGraphs () : (Int, (Int, Int)[])[] {
    return [(3, []),
            (4, [(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]),
            (5, [(4, 0), (2, 1), (3, 1), (3, 2)]),
            (5, [(0, 1), (1, 2), (1, 3), (3, 2), (4, 2), (3, 4)]),
            (5, [(0, 1), (0, 2), (0, 4), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)]),
            (6, [(0, 1), (0, 2), (0, 4), (0, 5), (1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (3, 4), (3, 5), (4, 5)])];
    // Graphs with 6+ vertices can take several minutes to be processed;
    // in the interest of keeping test runtime reasonable we're limiting most of the testing to graphs with 5 vertices or fewer.
}

    operation ColorEqualityOracle_Nbit_Reference (c0 : Qubit[], c1 : Qubit[], target : Qubit) : Unit is Adj+Ctl {
    within {
        for (q0, q1) in Zipped(c0, c1) {
            // compute XOR of q0 and q1 in place (storing it in q1)
            CNOT(q0, q1);
        }
    }
    apply {
        // if all XORs are 0, the bit strings are equal
        (ControlledOnInt(0, X))(c1, target);
    }
}

operation VertexColoringOracle_Reference (V : Int, edges : (Int, Int)[], colorsRegister : Qubit[], target : Qubit) : Unit is Adj+Ctl {
    let nEdges = Length(edges);
    use conflictQubits = Qubit[nEdges];
    within {
        for ((start, end), conflictQubit) in Zipped(edges, conflictQubits) {
            // Check that endpoints of the edge have different colors:
            // apply ColorEqualityOracle_Nbit_Reference oracle; if the colors are the same the result will be 1, indicating a conflict
            ColorEqualityOracle_Nbit_Reference(colorsRegister[start * 2 .. start * 2 + 1],
                                               colorsRegister[end * 2 .. end * 2 + 1], conflictQubit);
        }
    }
    apply {
        // If there are no conflicts (all qubits are in 0 state), the vertex coloring is valid
        (ControlledOnInt(0, X))(conflictQubits, target);
    }
}
```

```
48
49      function IsVertexColoringValid_Reference (V : Int, edges: (Int, Int)[], colors: Int[]) : Bool {
50          for (start, end) in edges {
51              if colors[start] == colors[end] {
52                  return false;
53              }
54          }
55          return true;
56      }
57
58      @Test("QuantumSimulator")
59      operation T23_GroversAlgorithm () : Unit {
60          for (V, edges) in ExampleGraphs() {
61              Message($"Running on graph V = {V}, edges = {edges}");
62              let coloring = GroversAlgorithm(V, VertexColoringOracle_Reference(V, edges, _, _));
63              Fact(IsVertexColoringValid_Reference(V, edges, coloring),
64                   $"Got incorrect coloring {coloring}");
65              Message($"Got correct coloring {coloring}");
66      }
```

Listing 4. 图着色问题测试结果

```
1  Starting test execution, please wait...
2  A total of 1 test files matched the specified pattern.
3  Running on graph V = 3, edges = []
4  Testing V = 3, edges = []
5  Trying iteration 1
6  Testing V = 4, edges = [(0, 1),(0, 2),(0, 3),(1, 2),(1, 3),(2, 3)]
7  Got correct coloring [0,0,1]
8  Running on graph V = 4, edges = [(0, 1),(0, 2),(0, 3),(1, 2),(1, 3),(2, 3)]
9  Trying iteration 1
10 Trying iteration 2
11 Got correct coloring [2,0,3,1]
12 Running on graph V = 5, edges = [(4, 0),(2, 1),(3, 1),(3, 2)]
13 Trying iteration 1
14 Got correct coloring [0,2,1,3,3]
15 Running on graph V = 5, edges = [(0, 1),(1, 2),(1, 3),(3, 2),(4, 2),(3, 4)]
16 Trying iteration 1
17 Got correct coloring [1,2,0,3,1]
18 Running on graph V = 5, edges = [(0, 1),(0, 2),(0, 4),(1, 2),(1, 3),(2, 3),(2, 4),(3, 4)]
19 Trying iteration 1
20 Got correct coloring [2,0,3,2,0]
21 Running on graph V = 6, edges = [(0, 1),(0, 2),(0, 4),(0, 5),(1, 2),(1, 3),(1, 5),(2, 3),(2, 4),(3, 4),(3, 5),(4,
       5)]
22 Trying iteration 1
23 Testing V = 5, edges = [(4, 0),(2, 1),(3, 1),(3, 2)]
24 Testing V = 5, edges = [(0, 1),(1, 2),(1, 3),(3, 2),(4, 2),(3, 4)]
25 Testing V = 5, edges = [(0, 1),(0, 2),(0, 4),(1, 2),(1, 3),(2, 3),(2, 4),(3, 4)]
26 Got correct coloring [3,2,1,3,2,0]
```