南開大學
Nankai University

计算机学院

算法导论大作业

基于动态规划算法的强化学习

姓名：张铭徐

学号：2113615

专业：计算机科学与技术

2023 年 5 月 28 日

目录

1 摘要

动态规划是一种从局部最优解逐渐拓展状态，通过状态转移方程得到最优解的算法，其具有时间复杂度低，求解的解具有最优性等特点，在众多领域都有着极为重要的应用。本文将从机器学习中的强化学习角度，详细讨论动态规划算法在强化学习中的应用，并给出一个较为简单的基于贝尔曼方程的强化学习的例子————机器人自动走迷宫，我们将同时给出基于搜索算法的迷宫案例和强化学习的迷宫案例供参考。

关键词：动态规划，强化学习，贝尔曼方程，深度优先搜索

2 背景知识介绍

2.1 动态规划算法介绍

动态规划（Dynamic Programming）是一种解决多阶段决策过程优化问题的算法思想。其算法思想是通过将问题划分为多个重叠的子问题，并且按照一定的顺序求解子问题，最终得到原问题的最优解。

动态规划算法通常适用于满足最优子结构和重叠子问题性质的问题。最优子结构指的是一个问题的最优解包含其子问题的最优解。重叠子问题指的是子问题之间存在重复计算的情况。

动态规划算法的基本思想是：通过存储子问题的解来避免重复计算，并且通过已解决的子问题的解构建更大规模问题的解。一般情况下，动态规划算法包含以下步骤：

- 确定状态：将原问题划分为若干个子问题，确定问题的状态，并定义状态表示。
- 确定状态转移方程：通过观察问题的特点，得到问题状态之间的转移关系。即确定问题的递推关系，用数学公式表示出来。
- 确定边界条件：确定最简单的子问题的解，也就是边界条件。
- 确定计算顺序：确定计算子问题的顺序，通常是自底向上的顺序，从边界条件开始逐步计算，直到得到原问题的解。
- 计算最优解：按照确定的计算顺序，通过状态转移方程计算每个子问题的解，并将其存储下来，以便后续使用。最终得到原问题的最优解。

我们考虑一个经典的背包问题以更加深刻的理解动态规划算法：背包问题是指有一个背包容量为 C，有 n 个物品，每个物品有自己的重量 w 和价值 v，目标是找到一种装法，使得装入背包的物品总价值最大。显然，对于贪心算法，我们装相对重量最大的物品进入背包的是错误的，没有考虑到背包的空余等因素，所以对于这个问题，我们考虑使用动态规划算法解决，我们仿照刚刚给出的求解算法的范式，其步骤如下：

- 确定状态：设 dp[i][j] 表示前 i 个物品放入容量为 j 的背包中所能获得的最大价值。
- 确定状态转移方程：对于第 i 个物品，可以选择放入背包或不放入背包。如果选择放入背包，那么背包的容量减少 j-w[i]，对应的价值增加 v[i]；如果选择不放入背包，背包的容量保持不变，对应的价值不增加。因此，状态转移方程为：$dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])$
- 确定边界条件：dp[0][j] = 0，表示前 0 个物品放入任何容量的背包中所能获得的最大价值都为 0；dp[i][0] = 0，表示容量为 0 的背包中所能获得的最大价值都为 0。

- 确定计算顺序：由于状态转移方程中的 dp[i][j] 依赖于 dp[i-1][j] 和 dp[i-1][j-w[i]]，因此可以选择自底向上的计算顺序，先计算较小规模的子问题，再逐步计算较大规模的子问题。
- 计算最优解：根据确定的计算顺序，按照状态转移方程计算每个子问题的解，最终得到 dp[n][C]，即为背包问题的最优解。

当然，对于上面的例子，我们还可以使用滚动数组等方法进行优化，将其中一个维度压缩掉，由于本文讨论的重点不在于此，于是在这里不给出滚动数组优化空间复杂度的例子。以上便是我们动态规划的算法介绍，接下来我们考虑强化学习。

2.2 强化学习知识介绍

强化学习 [2] 是机器学习中重要的学习方法之一，与监督学习和非监督学习不同，强化学习并不依赖于数据，并不是数据驱动的学习方法，其旨在与发挥智能体 (Agent) 的主观能动性，在当前的状态 (state) 下，通过与环境的交互，通过对应的策略，采用对应的行动 (action)，获得一定的奖赏 (reward)，通过奖赏来决定自己下一步的状态。

强化学习的几个重要的组分是：

- 环境，即智能体所处的外来环境，环境可以提供给智能体对应的状态信息，并且基于智能体一定的奖赏或者乘法。
- 智能体：智能体是强化学习中的学习和决策主体，他可以通过与环境的交互来学习改进其在当前环境下采取的决策策略。
- 状态：用于描述当前环境与智能体的情况，或者所观测的具体的变量。
- 行动：智能体基于观察到的状态所采取的操作或者决策。
- 奖励：环境根据智能体所采取的行动和当前的状态给与其的反馈。

强化学习的目标是通过学习，不断优化自己的决策策略，使得对于任意一个状态下，智能体所采取的行动，都能够获得最大的奖励，值得注意的是，这种奖励是长期奖励，而非眼前奖励。我们为了实现这一目标，通常会引入价值函数和动作-价值函数来评估行为的好坏，同时使用一定的算法和方法来不断迭代价值函数，在优化迭代价值函数同时，我们实际上就是在优化智能体的决策策略，这是因为事实上价值函数就是表示智能体在时刻 t 处于状态 s 时，按照策略 采取行动时所获得回报的期望。价值函数衡量了某个状态的好坏程度，反映了智能体从当前状态转移到该状态时能够为目标完成带来多大“好处”。我们通过迭代价值函数，就可以不断的迭代自己的行动策略 。

我们接下来介绍强化学习的一些基础知识：

2.2.1 离散马尔科夫过程

马尔科夫过程是一种具有马尔科夫性质的随机过程; 马尔科夫性质指的是在给定当前状态的情况下，未来状态的概率分布仅依赖于当前状态，而与过去状态无关。这意味着系统的未来行为只与当前状态有关，与之前经历的状态序列无关，即并不是一个时间序列。

马尔科夫过程由四个主要元素组成：

- 状态集合：马尔科夫过程中可能的状态的集合。

- 状态转移概率：描述在给定当前状态的情况下，转移到下一个状态的概率分布。通常用转移矩阵表示，其中每个元素表示从一个状态到另一个状态的概率。矩阵类似于图论中的邻接矩阵，$m[i][j]$ 代表从状态 i 转移到状态 j 的概率。
- 奖励函数：用于描述在状态转移过程中的奖励或反馈信号。它指定了在从一个状态转移到另一个状态时，系统获得的即时奖励。
- 初始状态分布：描述系统在开始时所处的状态的概率分布。

在马尔科夫过程中，我们可以使用马尔科夫链来描述系统在一系列离散时间步骤中的状态转移。马尔科夫链是马尔科夫过程的一种特殊情况，它是一个离散时间、离散状态的随机过程。在马尔科夫链中，每个时间步骤系统处于一个特定的状态，并且根据状态转移概率从一个状态转移到下一个状态。

马尔科夫链的特点是马尔科夫性，即当前状态到未来状态的转移仅依赖于当前状态，而与过去状态无关。这种特性使得马尔科夫链具有无记忆的性质。马尔科夫链可以用一个状态转移矩阵来表示，矩阵中的元素表示从一个状态转移到另一个状态的概率。状态转移矩阵是一个方阵，其中每一行表示当前状态，每一列表示下一个状态，矩阵中的元素表示对应状态转移的概率。

马尔科夫链的一个重要性质是平稳分布即当系统经过长时间的状态转移后，状态分布达到一个稳定的分布。平稳分布是状态转移矩阵的一个特征向量，其对应的特征值为 1。

在强化学习中，马尔科夫过程和马尔科夫链提供了对环境和状态转移的建模方法。由于马尔科夫性，我们可以对强化学习的一系列内容进行对应的数学建模：我们首先定义马尔科夫的奖励过程：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

其中 G_t 表示时间步 t 的折扣回报，$R_{t+1}, R_{t+2}, R_{t+3}, \dots$ 是后续时间步骤中获得的奖励，γ 是折扣因子，它决定了未来奖励相对于即时奖励的重要性。

接下来我们考虑学习策略：一个好的策略是在当前状态下采取了一个行动后，该行动能够在未来收到最大化的反馈 G_t，而为了对策略函数 进行评估，我们首先需要定义价值函数和动作价值函数：

价值函数：

$$V : S \to \mathbb{R}$$

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

在这个公式中，V 表示价值函数，它将状态 s 映射到一个实数值。$V_\pi(s)$ 表示在第 t 步状态为 s 时，按照策略 π 行动后，在未来所获得反馈值的期望。

动作-价值函数：

$$q : S \times A \to \mathbb{R}$$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

在这个公式中，q 表示动作-价值函数，它将状态 s 和动作 a 映射到一个实数值。$q_\pi(s, a)$ 表示在第 t 步状态为 s 时，按照策略 π 采取动作 a 后，在未来所获得反馈值的期望。

我们强化学习的目标是找到一个最佳的策略 ，使得对于任意一个状态 s，根据策略所选择的动作 a 都能得到最大的价值 G，那么我们便可以从优化策略变成优化动作价值函数和价值函数。

2.2.2 贝尔曼方程

贝尔曼方程 (Bellman Equation)[1] 也被称作动态规划方程，由理查德 · 贝尔曼提出。

根据定义，状态价值函数表示在当前状态 s 下按照策略 π 行动后所获得的期望回报（即累积奖励）：

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

其中，G_t 表示从时间步 t 开始未来所有的折扣累积奖励。我们可以将 G_t 拆分为当前时间步的即时奖励 R_{t+1} 和从下一个时间步 $t+1$ 开始的累积奖励：

$$G_t = R_{t+1} + \gamma G_{t+1}$$

其中，γ 是折扣因子，用于衡量未来奖励的重要性。

根据马尔科夫性质，当前状态 S_t 的价值函数 $V_\pi(S_t)$ 可以用下一个状态 S_{t+1} 的价值函数 $V_\pi(S_{t+1})$ 表示：

$$V_\pi(S_t) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s]$$

进一步展开，我们可以使用动作 A_t 和下一个状态 S_{t+1} 的联合分布表示：

$$V_\pi(S_t) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R_{t+1} + \gamma V_\pi(S_{t+1})]$$

其中，$\pi(a|s)$ 表示在状态 s 下选择动作 a 的概率，$P(s'|s,a)$ 表示从状态 s 采取动作 a 转移到下一个状态 s' 的概率。

将上述表达式中的累积奖励 $V_\pi(S_{t+1})$ 替换为对下一个状态所有可能动作的价值函数的期望值 $V_\pi(S_{t+1})$，得到最终的贝尔曼方程：

$$V_\pi(s) = \sum_a \pi(a|s) \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) V_\pi(s') \right)$$

其中，$r(s,a)$ 表示在状态 s 采取动作 a 后获得的即时奖励。

动作价值函数（Action-Value Function）的贝尔曼方程的推导过程与状态价值函数类似，只是在计算累积奖励时需要考虑采取的动作：

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

将累积奖励拆分为即时奖励和从下一个时间步开始的累积奖励：

$$G_t = R_{t+1} + \gamma G_{t+1}$$

根据马尔科夫性质，当前状态动作对 $Q_\pi(S_t, A_t)$ 可以用下一个状态 S_{t+1} 和下一个动作 A_{t+1} 的价值函数 $Q_\pi(S_{t+1}, A_{t+1})$ 表示：

$$Q_\pi(S_t, A_t) = \mathbb{E}_\pi[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

展开并使用联合分布，得到贝尔曼方程：

$$Q_\pi(s,a) = \sum_{s'} P(s'|s,a) \left(r(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s',a') \right)$$

其中，$r(s,a,s')$ 表示在状态 s 采取动作 a 后转移到状态 s' 并获得的即时奖励，$P(s'|s,a)$ 表示从状态 s 采取动作 a 转移到下一个状态 s' 的概率。

贝尔曼方程描述了价值函数或动作-价值函数的递推关系，是研究强化学习问题的重要手段。其中价值函数的贝尔曼方程描述了当前状态价值函数和其后续状态价值函数之间的关系，即当前状态价值函数等于瞬时奖励的期望加上后续状态的 (折扣) 价值函数的期望。而动作-价值函数的贝尔曼方程描述了当前动作-价值函数和其后续动作-价值函数之间的关系，即当前状态下的动作-价值函数等于瞬时奖励的期望加上后续状态的 (折扣) 动作-价值函数的期望。

在实际中，需要计算得到最优策略以指导智能体在当前状态如何选择一个可获得最大回报的动作。求解最优策略的一种方法就是去求解最优的价值函数或最优的动作-价值函数 (即基于价值方法)。一旦找到了最优的价值函数或动作-价值函数，自然而然也就是找到最优策略。当然，在强化学习中还有基于策略和基于模型等不同方法，在本文中，我们仅讨论值优化策略。

在这里我们给出值迭代和策略迭代的一般过程.

对于值迭代：

- 初始化所有状态的值函数为任意值。
- 对于每个状态，根据当前值函数和当前策略，计算其更新后的值函数。
- 重复步骤 2，直到值函数收敛到最优值函数。
- 根据最优值函数，得到最优策略.

而策略迭代是一种同时优化价值函数和策略的方法。它交替进行策略评估和策略改进的步骤，直到策略收敛到最优策略。具体而言，策略迭代的步骤如下：

- 初始化策略为任意策略。
- 策略评估：根据当前策略计算每个状态的值函数。
- 策略改进：根据当前值函数选择每个状态的最优行动，更新策略。
- 重复步骤 2 和步骤 3，直到策略收敛到最优策略。

值迭代和策略迭代都可以用于找到最优的策略，但它们在算法的执行方式和收敛性质上有所不同。值迭代通常需要更多的迭代次数来达到收敛，但每次迭代的计算量较小。策略迭代在每次迭代中都需要进行策略评估和策略改进的步骤，计算量较大，但通常收敛更快。

2.3 基于价值的策略优化与评估

我们考虑基于价值的策略优化与评估的方法，常见的方法有蒙特卡洛采用，动态规划和时序差分（Q-Learning）三种，在本文中，我们仅讨论其中的两种方法。

2.3.1 动态规划法进行策略优化

策略评估的目标是计算给定策略下的状态值函数，即评估当前策略在每个状态下的价值。动态规划策略评估方法基于贝尔曼方程来更新状态值函数，其中贝尔曼方程描述了状态值函数之间的关系。

动态规划策略评估的过程如下：

- 初始化：将所有状态的初始值函数估计为 0 或其他初始值。

- 迭代更新：重复以下步骤直到收敛：

  a. 对于每个状态 s，计算其新的值函数 V(s)：$V(s) = \sum[p(s'|s,a) * (r(s,a,s') + \gamma * V(s'))]$ 其中，$P(s'|s,a)$ 是在状态 s 采取动作 a 后转移到状态 s' 的概率，$r(s,a,s')$ 是在状态 s 采取动作 a 后转移到状态 s' 时获得的即时奖励，γ 是折扣因子（用于权衡当前奖励和未来回报的重要性），$V(s')$ 是下一个状态 s' 的值函数。

  b. 对于所有状态 s，更新值函数 V(s) 为新的估计值。

- 收敛判断：检查值函数的变化是否小于预定义的阈值，如果满足收敛条件，则停止迭代。

Algorithm 1 基于动态规划的价值函数更新

```
procedure 价值函数更新
    repeat
        Δ ← 0
        for all s ∈ S do
            v ← V(s)
            V(s) ← Σ_{a∈A} π(s,a) Σ_{s'∈S} Pr(s' | s,a)[R(s,a,s') + γV(s')]
            Δ ← max(Δ, |v − V(s)|)
        end for
    until Δ < ϵ
end procedure
```

通过迭代更新状态值函数，动态规划策略评估方法可以逐步逼近最优状态值函数，即策略下每个状态的最大回报，针对上述过程，我们给出算法流程图如1所示。需要注意的是，动态规划策略评估方法要求完全了解环境的转移概率和奖励函数，因为它依赖于贝尔曼方程的精确计算。这在一些简单的问题中可能可行，但对于大型问题或连续状态空间的问题，这种方法往往是不可行的。在这些情况下，可以使用近似方法，如蒙特卡洛方法或基于函数逼近的方法（如值函数近似）来评估策略，于是我们引出了下面的方法，使用 Q-Learning 进行策略优化。

2.3.2 Q-Learning 进行策略优化

虽然算法名称是 Q-Learning，但是其本质仍然是基于动态规划的一种优化算法；Q-Learning 对于值函数的更新方式仍然是一个状态转移方程。在我看来，Q-Learning 似乎比我们上文所讨论的基于动态规划的价值函数更新更加类似于动态规划一些，下面我们将详细的介绍 Q-Learning 的相关内容。

Q-Learning 的本质是为了维护一个 Q 表,其中 Q 表存储了每一对可能的 (s_i, a_j) 所对应的 $q(s_i, a_j)$，即 Q 表中存储了智能体所有可能的状态和采取的行动所获得的奖赏。当我们维护好 Q 表后，对于任意一个状态 s_i，由于我们的强化学习是一个马尔科夫过程，获得的奖赏仅与状态有关而与时间无关，所以我们可以直接在 Q 表中找到对应行或者列，然后采用该行或该列中使 Q 值最大的举动 a_j，下表即一个 Q 表的示例。

$$Q = \begin{array}{c|ccc} & \text{动作}a_1 & \text{动作}a_2 & \text{动作}a_3 \\ \hline \text{状态}s_1 & q_{s_1,a_1} & q_{s_1,a_2} & q_{s_1,a_3} \\ \text{状态}s_2 & q_{s_2,a_1} & q_{s_2,a_2} & q_{s_2,a_3} \\ \text{状态}s_3 & q_{s_3,a_1} & q_{s_3,a_2} & q_{s_3,a_3} \end{array}$$

接下来我们考虑如何维护 Q 表：维护 Q 表是通过不断更新 Q 值实现的。在其中，Q 值的计算公式为：$Q(s_t, a) = R_{t+1} + \gamma \times \max_a Q(a, s_{t+1})$。具体来说，我们对于每一个时间步的状态，会选择一个

动作 a，值得注意的是，该动作的选择并不是通过简单的贪心找最大实现的，我们通常会使用 ϵ 贪心策略，即在某一个状态下，有 ϵ 的概率选择其余的部分进行探索，有 $1-\epsilon$ 的概率选择最高的奖赏进行迭代，ϵ 是超参数，为了平衡利用和探索，类似于 MCT 中的超参数 c。在选择完动作 a 后，观察到奖励和下一个状态，根据我们的更新规则对 Q 表的 Q 值进行更新，然后跳转到下一个状态，以此类推。而我们的更新规则是引入了一个松弛变量：$Q(s_t,a)=(1-\alpha)\times Q(s_t,a)+\alpha\times(R_{t+1}+\gamma\times\max_a Q(a,s_{t+1}))$。也就是说，我们的值并不完全通过计算的新 Q 值得到，而是通过新 Q 值和旧 Q 值线性加权得到的，通过这种操作，可以让我们 Q 表的更新过程变得平稳，曲线更加平滑，迭代收敛的效果更好。上述过程我们整理出了算法流程图如2所示

Algorithm 2 Q-Learning 算法

初始化 $Q(s,a)$，对于所有的 $s\in S$，$a\in A$，任意赋值
初始化当前状态 s
repeat
    根据探索策略（例如，ϵ-贪心策略）选择动作 a
    执行动作 a，观察奖励 r 和下一个状态 s'
    使用 Q-Learning 更新规则更新 $Q(s,a)$：
        $Q(s,a)\leftarrow Q(s,a)+\alpha\left(r+\gamma\max_{a'}Q(s',a')-Q(s,a)\right)$
    将 s 更新为 s'
until 满足终止条件

3 情景描述及问题建模

3.1 情景描述

接下来我们给出一个实际问题：自动走迷宫。假设有一个机器人处在一个迷宫的起点，迷宫可以看做是一个 $n*m$ 的矩阵 M，对于该迷宫的任意一个位置 $M(i,j)$，机器人都有四种行动集合，分别是向上，向左，向右，向下走，但是既然是迷宫，就有可能有屏障的存在，即可能相邻位置不存在通路，而是一堵墙，如果采取对应的行动无法顺利的转移状态。我们期望于给定一个迷宫，机器人可以自动的找到从起点到终点的路径，我们给出了一个示意图3.1：

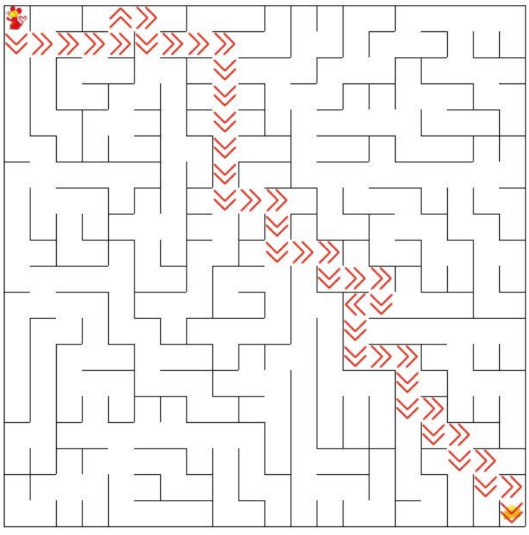

图 3.1: 机器人走迷宫示意图

3.2 问题建模与算法设计

该是一个经典的强化学习的基本问题，我们可以将机器人看做是智能体，将迷宫看做环境，将机器人所处的地方视为状态，机器人的行动称之为动作，机器人根据当前的迷宫周围的环境所采取的动作称之为策略，每次移动是否合法，是否到达迷宫终点视为环境给予的奖励。

3.2.1 深度优先搜索算法

我们考察机器人走迷宫问题，由于迷宫是有限范围的迷宫，迷宫的状态是有限的，针对这种情况，我们可以采用深度优先搜素或宽度优先搜索实现对状态的穷举，然后探索出一条符合条件的路径。针对宽度优先搜索，我们可以构建一颗搜索树，然后通过层次遍历的办法来搜索，我们在这里考虑使用深度优先搜索。

深度优先搜索的实现方法是递归与堆栈，可以看做“一条路走到黑，不见黄河不回头，不见棺材不落泪”，会沿着一条路径一直走，直到走到不合法死胡同为止。我们在当前状态下，可以通过函数得到可以走的路径，然后从中选择一个，进行递归传参，然后继续这个过程，直到走到死胡同或者路径终点为止。

不过值得注意的是，我们需要用到回溯算法：回溯算法的本质是，我们这次搜索不希望对于下一次搜索产生影响，所以想要完全消除掉本次搜索对于环境的影响，在这里我们采用了一个 vis 数组来判定该状态是否被遍历过，采用了一个 path 数组来存储每次采取的动作是什么，我们回溯的思路便是在搜索完或者走到死胡同后往回走，将当前搜索完的 path 删掉，同时 vis 数组置 0。

这种算法的正确性是显然的，并不会陷入无穷的遍历过程，因为对于每一个状态，我们采取的动作都是唯一的，我们在每一个状态都会遍历对应的动作，而不会往回遍历，回溯的目的并不是回溯动作，而是删除当前搜索对于未来搜索的影响。这便是我们深度优先搜索的设计思想。我们对于设计思想，给出算法流程图如下3所示：

Algorithm 3 机器人走迷宫的递归实现

```
1:  procedure DFS(location)
2:      if location == destination then
3:          return path
4:      end if
5:      for move in can_move do
6:          计算出采取该行动后的 new_location
7:          if is_visit_m[new_location] == 0 then
8:              path.append(move)
9:              result ← dfs(new_location)
10:             if result is not None then
11:                 return result
12:             end if
13:         end if
14:     end for
15:     is_visit_m[location] ← 0
16:     path.pop()
17: end procedure
```

接下来我们对时间复杂度进行分析：要分析该算法的时间复杂度，需要考虑每个步骤的时间消耗和重复执行的次数。

首先，我们来看主要的递归函数 dfs。在每次调用 dfs 时，会进行一些基本的操作，如标记当前位置为已访问，将当前位置坐标记录到路径数组中，并检查是否到达终点。这些基本操作的时间复杂度

可以看作是常数时间，即 O(1)。

接下来，我们考虑 for 循环部分。这个循环会遍历每个邻居节点，并递归调用 dfs 函数。在最坏情况下，每个邻居节点都需要递归探索，而每次递归调用 dfs 都会继续向下递归，直到找到终点或无法继续移动。因此，每个节点最多会被遍历一次。

对于每个节点，我们都需要检查其邻居节点的合法性并进行一些判断操作。这些操作的时间复杂度也可以看作是常数时间，即 O(1)。

因此，在最坏情况下，算法的时间复杂度可以表示为 O(V)，其中 V 是迷宫中的有效节点数。实际上，由于每个节点最多被遍历一次，可以认为时间复杂度是线性的，即 O(V)。

总结起来，给定迷宫大小为 sizesize，该算法的时间复杂度为 $O(size * size)$ 或简写为 $O(n^2)$，其中 n 为迷宫的尺寸。

3.2.2 强化学习 Q-Learning

根据我们上面对于 Q-Learning 的描述，这部分的算法设计已经很清晰了，直接给出对应的算法流程图4

Algorithm 4 机器人走迷宫的 Q-Learning 算法

procedure Q-Learning(maze, num_episodes, learning_rate, discount_factor, exploration_rate)
    初始化 Q 表：$Q \leftarrow$ zeros((num_states, num_actions))
    for *episode* in *num_episodes* **do**
        初始化状态：$state \leftarrow$ maze.reset_robot()
        while 未到达目标状态 **do**
            根据当前状态和 Q 表选择动作：$action \leftarrow$ epsilon_greedy($state$, $exploration\_rate$)
            执行动作并观测新状态和奖励：$new\_state, reward \leftarrow$ maze.move_robot($action$)
            更新 Q 表：$Q[state, action] \leftarrow (1 - learning\_rate) \cdot Q[state, action] + learning\_rate \cdot (reward + discount\_factor \cdot \max(Q[new\_state, :]))$
            更新当前状态：$state \leftarrow new\_state$
        end while
        降低探索率：$exploration\_rate \leftarrow exploration\_rate \cdot decay\_rate$
    end for
    return Q
end procedure

算法首先初始化 Q 表，并通过一定数量的训练周期进行学习。在每个周期中，机器人根据当前状态和 Q 表选择动作，并执行该动作来观测新的状态和奖励。然后，算法使用 Q-Learning 更新规则来更新 Q 表中对应状态和动作的值。最后，算法根据一定的探索率逐步降低探索行为，以便在训练过程中逐渐加强对现有知识的利用。最终，算法返回训练后得到的 Q 表。

对于时间复杂度：Q-Learning 算法的主要步骤如下：

- 初始化 Q 表：对于每个状态和动作的组合，初始化其对应的 Q 值为 0。
- 选择动作：根据当前状态和 Q 表选择下一步的动作。
- 执行动作并观察环境反馈：根据选定的动作与环境进行交互，并获得奖励和下一个状态。
- 更新 Q 值：使用 Q-Learning 算法更新 Q 表中当前状态和动作对应的 Q 值。
- 转移到下一个状态：将下一个状态作为当前状态，并返回第 2 步。

接下来，我们逐步分析每个步骤的时间复杂度。

- 初始化 Q 表：初始化 Q 表的时间复杂度为 $O(S*A)$，其中 S 是状态的数量，A 是动作的数量。
- 选择动作：在 Q 表中查找最大的 Q 值的时间复杂度为 $O(A)$。当使用 -greedy 策略时，还需要生成一个 0 到 1 之间的随机数，时间复杂度为 $O(1)$。
- 执行动作并观察环境反馈：执行动作和观察环境反馈的时间复杂度取决于具体的环境。在这里我们的复杂度为 $O(1)$。
- 更新 Q 值：更新 Q 值的时间复杂度为 O(1)，因为只需要执行一些基本的数学运算。
- 转移到下一个状态：转移到下一个状态的时间复杂度为 O(1)，因为只需要将下一个状态作为当前状态。

根据上述分析，可以得出 Q-Learning 算法的总体时间复杂度为 $O(S*A*N)$，其中 S 是状态的数量，A 是动作的数量，N 是训练的总步数（包括所有的训练周期）。

4 编程实现

我们实现了一个 Python 代码和一个 C++ 代码，由于 Python 有丰富多种的库函数可供我们选择，相对而言代码量比较小；而对于 C++，由于 C++ 没有提供给我们迷宫类的对应库函数，所以我们需要手写封装一个 Maze 类。

4.1 Python 部分

对于 Python 部分，我们在这里仅给出训练部分和测试过程中选择选择对应动作的代码，Python 部分如下所示：

Q-Learning 训练部分和测试部分代码

```python
    def train_update(self):
        self.state = self.sense_state()  # 获取机器人当初所处迷宫位置
        self.create_Qtable_line(self.state)  # 如果不存在则添加进入Q表
        action = random.choice(self.valid_action) if random.random() < self.epsilon
            else max(
             self.q_table[self.state], key=self.q_table[self.state].get)  # 选择动作
        reward = self.maze.move_robot(action)  # 以给定的动作（移动方向）移动机器人
        next_state = self.sense_state()  # 获取机器人执行动作后所处的位置
        self.create_Qtable_line(next_state)  #检索Q表，如果不存在则添加进入Q表
        self.update_Qtable(reward, action, next_state)  # 更新 Q 表 中 Q 值
        self.update_parameter()  # 更新其它参数
        return action, reward

    def test_update(self):
        self.state = self.sense_state()  # 获取机器人当初所处迷宫位置
        self.create_Qtable_line(self.state)  #
            对当前状态，检索Q表，如果不存在则添加进入Q表
        action = max(self.q_table[self.state],
                     key=self.q_table[self.state].get)  # 选择动作
```

```
        reward = self.maze.move_robot(action)  # 以给定的动作（移动方向）移动机器人
        return action, reward
```

4.2 C++ 部分

4.2.1 Q-Learning 部分

基于 C++ 的部分，我们首先实现了迷宫类，对应 Python 库中的 Maze，具体而言，代码如下：

迷宫类的实现

```
class Maze {
    std::vector<std::vector<int>> maze_data;
    std::pair<int, int> robot_position;

public:
    Maze(const std::vector<std::vector<int>>& data);
    std::vector<std::string> can_move_actions(const std::pair<int, int>& position);
    void reset_robot();
    std::string sense_robot();
    double move_robot(const std::string& action);
    void print_maze();
    std::pair<int, int> get_robot_position() const;  // 新增函数，获取机器人当前位置
};
```

我们介绍一下迷宫类的每一个成员函数的功能：

- 构造函数：初始化用户输入的迷宫。
- can_move_actions: 获取所有当前状态下所有合法的移动。
- reset_robot：重置机器人到初始位置。并重置 ϵ 为初始值
- sense_robot：获取机器人当前位置。
- move_robot：以一定的动作移动机器人，返回对应动作的奖赏。
- print_maze：输出迷宫。
- get_robot_position：获取机器人当前位置。

值得注意的是，我们在 move_robot 中，人为设定了对应的奖励，我们采取的奖励策略是：如果达到目的地，那么 $reward+ = 50$，如果撞墙，那么 $reward- = 10$，如果是其他情况，则 $reward- = 0.1$，对于最后一点，即其他情况我们也削减了奖赏，这样以来，可以鼓励智能体在迷宫中寻找尽可能短的路径以获得更大的奖赏。接下来我们给出基于 Q-Learning 实现的 QRobot 类：

QRobot 类的实现

```

class QRobot {
    Maze maze;
    std::pair<int, int> state;
```

```
    std::string action;
    double alpha;
    double gamma;
    double epsilon0;
    double epsilon;
    int t;
    std::vector<std::pair<std::pair<int, int>, std::string>> history;
    std::map<std::pair<int, int>, std::map<std::string, double>> q_table;
public:
    QRobot(const Maze& maze, double alpha = 0.5, double gamma = 0.8, double epsilon0
        = 0.99);
    std::vector<std::string> current_state_valid_actions();
    void reset();
    double update_parameter();
    std::string sense_state();
    void create_Qtable_line(const std::pair<int, int>& state);
    void update_Qtable(double r, const std::string& action, const std::pair<int,
        int>& next_state);
    std::pair<std::string, double> train_update();
    std::pair<std::string, double> test_update();
    std::vector<std::pair<int, int>> get_path();
    std::pair<int, int> get_robot_position() const {
    return maze.get_robot_position();
}
};
```

同样的，我们介绍一下类中的成员变量和成员函数的含义：

- maze: 一个 Maze 对象，表示机器人所在的迷宫。
- state: 一个 pair<int, int> 类型的变量，表示机器人当前的状态（位置）。
- action: 一个 string 类型的变量，表示机器人选择的动作。
- α: 学习速率，用于控制 Q 值的更新速度。
- γ: 折扣因子，用于权衡当前奖励和未来奖励的重要性。
- ϵ_0: 探索率的初始值。
- ϵ: 当前的探索率。
- t: 训练步数，用于调整探索率。
- history: 一个:vector<std::pair<std::pair<int, int>, std::string» 类型的变量，用于存储机器人的历史状态和动作。
- q_table: 一个 std::map<std::pair<int, int>, std::map<std::string, double» 类型的变量，表示 Q 值表，存储每个状态和动作的 Q 值。
- QRobot: 构造函数，用于初始化 QRobot 对象。接受一个 Maze 对象和可选的学习率、折扣因子和探索率初始值作为参数。

- current_state_valid_actions(): 返回机器人当前状态下可用的动作列表。
- reset(): 重置机器人的状态和探索率。
- update_parameter(): 更新探索率和训练步数。
- string sense_state(): 获取机器人当前的状态。
- create_Qtable_line(const std::pair<int, int> state): 创建 Q 值表中的新行。
- update_Qtable(double r, const std::string action, const std::pair<int, int> next_state): 更新 Q 值表中的 Q 值。
- pair<std::string, double> train_update(): 在训练过程中选择动作并更新 Q 值。
- pair<std::string, double> test_update(): 在测试过程中选择动作。
- vector<std::pair<int, int» get_path(): 获取机器人的路径。
- pair<int, int> get_robot_position() const: 获取机器人的位置。

更加具体的代码实现在这里就不再展示，完整的项目，包括算法的源码，可执行文件，Python 代码等都可以在算法导论仓库中找到。

4.2.2 DFS 深度优先搜索部分

对于 DFS 部分，基于我们上述的算法流程图3，我们直接给出对应的代码供参考：

深度优先搜索实现

```
bool visited[50][50]={0};
int counter=0;
int dfs_path_y[500];
int dfs_path_x[500];
bool dfs(int x, int y, const std::vector<std::vector<int>>& maze_data) {
    visited[y][x] = true;
    dfs_path_y[++counter]=y;
    dfs_path_x[counter]=x;
    if (maze_data[y][x] == 2) {
        return true;  // 找到了终点
    }
    std::vector<std::pair<int, int>> neighbors = {
        {x - 1, y},  // 左
        {x + 1, y},  // 右
        {x, y - 1},  // 上
        {x, y + 1}   // 下
    };
    for (const auto& neighbor : neighbors) {
        int nx = neighbor.first;
        int ny = neighbor.second;

        if (nx >= 0 && nx < maze_data[0].size() && ny >= 0 && ny < maze_data.size()
            && maze_data[ny][nx] != 1 && !visited[ny][nx]) {
```

```
            if (dfs(nx, ny, maze_data)) {
                return true;  // 找到了终点，返回true
            }
        }
    }
    counter--;  //回溯
    visited[y][x]=0;//回溯
    return false;  // 没有找到终点，返回false
}
```

5 结果展示

在本节中，我们将对搜索结果和强化学习的结果予以展示，我们首先先对问题进一步的加以描述，并对输入格式加以限制：

5.1 问题描述

机器人自动走迷宫问题是经典的搜索问题，由于其状态的有限性，我们可以通过多种搜索算法，例如启发式搜索，深度优先搜索，宽度优先搜索等实现；我们注意到，这几种典型的搜索算法找到一条路径容易，但是如果我想要找到一条最优的路径便显得不容易；搜索算法只能在解空间中返回其中一个解，而不能保证每次返回的都是最优解；基于此，我们考虑使用基于动态规划的强化学习方法来求最优解，采用 Q-Learning 的手段来解决这一问题。

5.2 输入格式

由于算法的规模限制，C++ 不便于处理十分复杂的强化学习，我们测试了规模为 10，20，30 等大小的迷宫，都可以完美的解决问题，在本例中，我们以 30 大小的为例。我们首先需要输入迷宫大小 $size$，意为迷宫是一个 $size * size$ 的矩阵，然后我们需要给出迷宫的样子，即输入 $size * size$ 个数，0 代表该位置可走，1 代表该位置不可走，2 代表终点，3 代表起点。不过值得注意的是，我们默认起点在 (0,0) 处，测试的时候请务必将起点设置为 (0,0), 不然深搜过程中有可能出现问题。

对于该任务，我们给出了 4 组测试样例，分别为大小为 10,15,20,30 的迷宫，对于更多的测试样例，可以自行构造，输入到文件中求解，不过对于较大规模的数据可能效果不会很好。

5.3 输出格式

对于强化学习部分，我们首先进行了 500 轮次的训练，并且将每 10 轮的训练结果进行输出，输出的结果为每一轮需要多少次能够到达目的地供参考，我们可以看到强化学习算法逐渐收敛到一个较好的水平上，该收敛的结果就是迷宫中从起点到终点的最短的路径。此外，我们还输出了训练后的结果，即从起点到终点的最短路径的走法，以及所需要的步数；最后，我们输出了基于深度优先搜索的步数和路径。

由于输出的内容较多，我们在这里只展示四张可视化的图，分别代表了规模为 10*10，15*15，20*20，30*30 即项目中附带的 4 个样例迷宫，强化学习和深度优先搜索所寻找到的路径，如下图5.6所示：

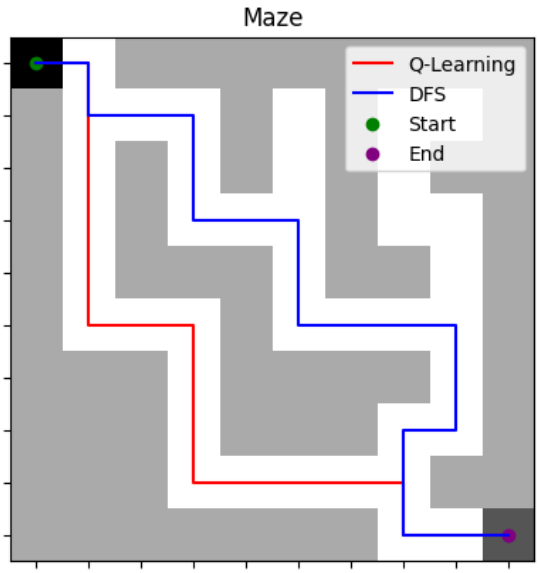

图 5.2: 10*10 迷宫

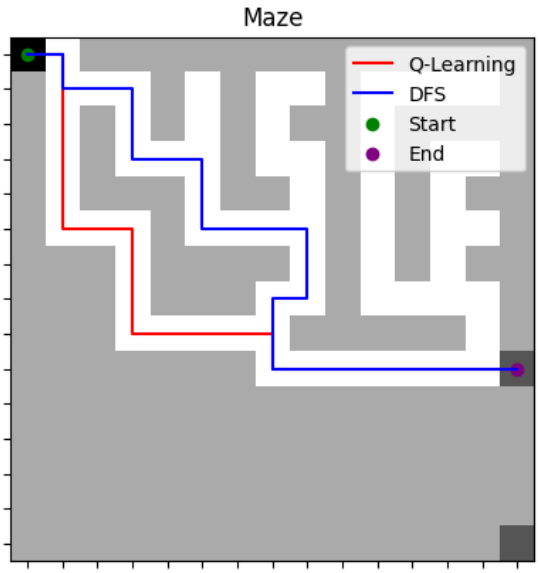

图 5.3: 15*15 迷宫

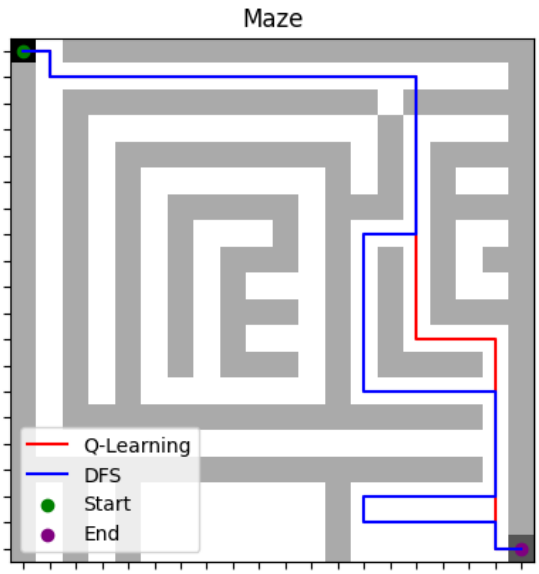

图 5.4: 20*20 迷宫

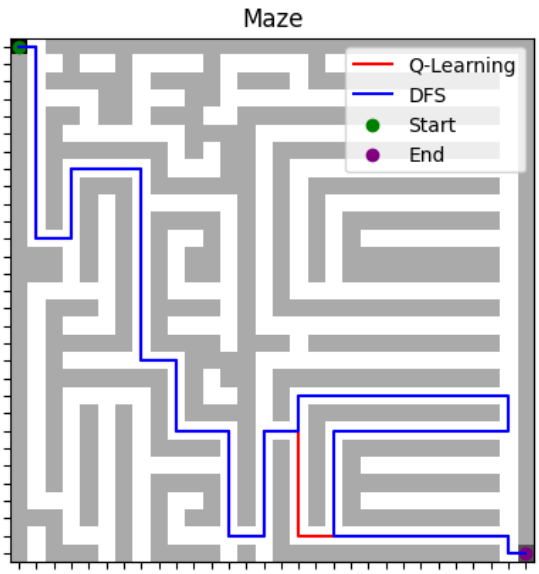

图 5.5: 30*30 迷宫

图 5.6: Q-Learning 和 DFS 算法迷宫路径图

5.4 结果分析

我们从给出的可视化图中尝试对结果进行一定的分析：我们首先可以看到 Q-Learning 算法经过若干次迭代后可以很好的找到迷宫中的最短路径，搜索算法只能在所有的解中寻求到一个，而我们的 Q-Learning 则可以稳定的找到最优的解，我们从图中明显可以看出，对于红线的路径显著的短于蓝色的线，这也成功的印证了我们上面所讨论的内容。

此外，对于 30*30 大小的迷宫，我们给出后几次训练的结果，如下表1所示：

Epochs	Steps
4971	78
4972	78
4973	78
4974	78
4975	80
4976	78
4977	78
4978	78
4979	78
4980	80
4981	80
4982	78
4983	78
4984	80
4985	80
4986	78
4987	78
4988	78

表 1: Epochs and Steps

我们可以看到，对于后几次训练而言，基本已经收敛，达到稳定状态，但是我们注意到，仍然有部分情况没有收敛到比较最优值，我们推测这是由于超参数和奖赏决定的：对于不同大小的迷宫，由于最优解的路径不同，我们需要对罚分进行一定的调整。

6 总结

本次大作业我们使用基于动态规划算法的 Q-Learning，在机器人走迷宫的例子上具体的应用了强化学习办法和深度优先搜索算法；并在前文中给出了 Q-Learning 的贝尔曼方程的推导过程。

我们发现：Q-Learning 算法在求解最优解问题上具有较为得天独厚的优势，但是同样存在一些问题，我们的 Q 值取决于动作价值函数，动作价值函数就与状态耦合，如果状态过多，有些状态可能始终无法采样到，因此对这些状态的 q 函数进行估计是很困难的，并且，当状态数量无限时，不可能用一张表 (数组) 来记录 q 函数的值。我们考虑多层感知机 (MLP) 是一个通用的函数逼近器，可以以任意精度逼近任意多项式函数，所以我们考虑采用深度神经网络来拟合 Q 函数，而不是通过值迭代的方法对 Q 值进行计算。这便是我们的 DQN(Deep-Q-Learning)。由于本文不探讨深度学习算法，故在这里不给出对应的说明。

当然，我们可以继续优化上面的内容代码，不断的调整超参数以最快的达到收敛点。

参考文献

[1] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.