Generated on 2025-06-02

ParaN3xus's Blog 2025

Archive of Blog posts in 2025

目录

2025-05-25

团精确计数的 Pivoter 算法

对 Pivoter 算法的介绍, 包括关键数据结构 SCT 的构建和性质, 以及如何使用 SCT 来计数团.

团精确计数的 Pivoter 算法 © 2025 by ParaN3xus is licensed under CC BY-NC-SA 4.0.

这是我“数据结构”课程大作业的一部分内容, 由于中文网络上似乎没有 Pivoter 算法的相关资料, 所以我整理成一篇文章来介绍这个算法.

1.1 任务

我所面临的任务是: 在给定的无向图中, 计算所有团(clique, 也即完全子图)的个数. 除此之外, 常见的任务还可以是计算某些点或者某些边参与的团的个数, 或者把这些团列出来.

方便起见, 下面默认任务是计算所有团的个数.

1.2 记号

为了防止混淆, 本文中递归树结构的边称为“连接”, 而图中的边称为“边”, 递归树结构的点称为“节点”, 而图中的点称为“顶点”.

记号	含义
$G(V, E)$	无向图, 其中 V 是顶点集, E 是边集
n	顶点数, 即 $\|V\|$
m	边数, 即 $\|E\|$
α	退化度, 也即图中最大团的大小
C	团, 也即完全子图
C_k	k 阶团, 也即包含 k 个顶点的团
$N(v)$	顶点 v 的邻居集
$N^+(v)$	顶点 v 的出邻居集, 对有向图或无向图的一个定向适用
$N(S, v)$	顶点集 S 中顶点 v 的邻居集, 即 $N(v) \cap S$
$\mathrm{SCT}(G)$	图 G 的简化团树, 即 Pivoter 算法的递归树
$\boldsymbol{T}$	SCT, 也即简化团树
$\mathfrak{h}$	SCT 中连接标签的 hold 类型
$\mathfrak{p}$	SCT 中连接标签的 pivot 类型
T	递归树中的一条路径, 也即从根到某个叶子节点的路径
$H(T)$	T 中标签类型为 $\mathfrak{h}$ 的所有边的标签中的节点的集合
$P(T)$	T 中标签类型为 $\mathfrak{p}$ 的所有边的标签中的节点的集合

表 1　符号表

1.3 Pivoter 算法

1.3.1 Pivoter 算法简介

Pivoter 算法是 Shweta Jain 等人于 2020 年提出的一种精确计数团的算法, 原论文发表在 WSDM 2020 上.

Pivoter 算法借鉴了 BK 算法的 Pivot 思想, 通过选择一个顶点作为 Pivot 构建一种具有良好性质的新的递归树结构 SCT(Succinct Clique Tree, 简化团树) 大大减少了朴素递归算法的时空代价. Pivoter 算法的时间复杂度是 $O(\alpha^2|\mathrm{SCT}(G)| + m + n)$, 空间复杂度是 $O(m + n)$.

1.3.2 Pivoter 算法的思想

1.3.2.1 朴素递归算法

我们可以先从朴素递归算法开始, 它的思路是: 所有包含 v 的团都可以通过把 v 添加到另一个团中得到, 后者所提到的团必定在 $N(v)$ 中.

所以我们可以通过下面的这个递归过程找到所有团:

```
GetCliques(V)
    If |V| ≤ 1:
        Return [[v]]
    R ← []
    For v in V:
        ℂ ← GetCliques(N(v))
        ℂ ← ℂ ∪ {C ∪ {v} | C ∈ ℂ}
        R ← R ∪ ℂ
    Return R
```

我们可以想到朴素递归算法的递归树:

- 每个节点都对应一个 V 的子集 S, 大多数时候是某个顶点的邻居, 我们称之为节点的标签
- 标签为 S 的节点的出连接对应了所有 $s \in S$, 我们称之为连接的标签

于是我们可以看出, 从根路径出发向下的任何路径(可以不到叶子节点)都给出一个团, 也即这条路径途径的所有连接的标签, 而且我们能通过这种方式获得所有团. 这是因为

1. 一个标签为 v 的节点的子树中的所有节点(包括子节点, 子节点的子节点, ...)显然都在 $N(v)$ 中, 也即他们都和 v 有边. 对路径上的所有节点都应用这个观察即可得知这是一个团
2. 任取一个团 $v_1, v_2, v_3 ... v_n$, 显然 v_1 标记的连接必定在递归树的根节点之下, 而 $v_2 ... v_n$ 都是 v_1 的邻居, 他们必定在这一连接所连接的另一个节点的标签中, 然后便可以反复应用这一规则, 直到说明所有 n 个节点都在树上, 而且呈一条向下的路径排列

但是显然这种方式会导致重复生成团, 一种避免的方式是对这个图生成一个有向无环图(DAG), 然后把 N 修改为只给出出邻居节点.

但是即使如此, 在大型图上我们也不能完整地构建这个递归树. 所以 Pivoter 算法试图找出一种压缩这棵树, 而且获得所有团的唯一表示的方式.

1.3.2.2 Pivoter 和 SCT

对于上述递归树中的一个标签为 S 的节点, 我们取出一个轴节点 $p \in S$, 于是 S 中的任一团 C 可以被分为三类

1. $p \in C$

2. $C \subset N(p)$
3. C 中有一个 p 的非邻居

前两类之间有一一对应的关系: 把每个第一类团里面的 p 拿掉就可以得到第二类团, 反之亦然, 于是我们这里可以只在树中继续处理第二类团.

对于这个节点, 我们递归调用以获取在 $N(p) \cap S$ 和 $N(u) \cap S$ 中的团, 其中 u 是 S 中 p 的非邻居节点. 这两种团分别对应前面所说的第二类和第三类.

这里 p 可以取度最高的节点, 这是因为节点的度越大, 他就更有可能在更多团中, 那我们省下的第一类节点的代价就越多.

按这种算法构造的递归树就是所谓 SCT. SCT 可以轻易在千万级的图上构建.

1.3.2.3 使用 SCT 进行团计数

和一般递归算法的递归树相比, SCT 的每条路径 T 的连接标签集仍然对应一个团, 但是不是所有团都有路径对应, 实际上, 这也是为什么一般递归算法的递归树如此巨大.

但是如果这样, 我们又如何使用 SCT 来计数所有团呢? 实际上这正是 Pivoter 算法的关键贡献所在: SCT 有一种良好的性质, 使得我们可以在图中找到所有团的唯一表示. 而我们正是通过这种方式来计数所有团的. 至于具体如何, 我们将在接下来的内容中介绍.

1.3.3 Pivoter 算法的实现

1.3.3.1 SCT 的构建

在此之前, 我们要先具体定义 SCT 的结构:

- SCT 是一棵树.
- SCT 的每个节点都有一个标签, 这个标签是一个顶点集. 其中根节点的标签是 V.
- SCT 的每个连接都有一个标签, 这个标签是一个顶点-类型对, 也即 $(v, \cdot)$, 其中 v 是该连接接近根节点一端的节点标签中的一个元素, 类型为 $\mathfrak{p}$ 或者 $\mathfrak{h}$.

下面我们给出一个 SCT 的构建算法, 这是一个 BFS 算法.

BuildSCT(G):
  $N^+ \leftarrow$ **BuildDAG**(G)
  Init $\boldsymbol{T}$ with root node R labeled by V
  $Q \leftarrow$ empty queue
  For v **in** V:
    $\mathcal{N} \leftarrow$ new node labeled by $N^+(v)$
    $\boldsymbol{T} \leftarrow \boldsymbol{T}$ with link labeled by $(v, \mathfrak{h})$ from R to v
    Push $\mathcal{N}$ into Q
  While Q is not empty:
    $\mathcal{P} \leftarrow$ pop node from Q
    $S \leftarrow$ label of $\mathcal{P}$
    If S is $\varnothing$:
      Continue
    $p \leftarrow \arg\max_p |N(S, p)|$
    $\mathcal{N} \leftarrow$ new node labeled by $N(S, p)$
    $\boldsymbol{T} \leftarrow \boldsymbol{T}$ with link labeled by $(p, \mathfrak{p})$ from $\mathcal{P}$ to $\mathcal{N}$
    Push $\mathcal{N}$ into Q

$\{v_1, v_2, ..., v_l\} \leftarrow S \setminus (p \cup N(p))$
For $i < l$:
  $\mathcal{N}_2 \leftarrow$ new node labeled by $N(S, v_i) \setminus \{v_1, v_2, ..., v_{i-1}\}$
  $\boldsymbol{T} \leftarrow \boldsymbol{T}$ with link labeled by $(v_i, \mathfrak{h})$ from $\mathcal{N}$ to $\mathcal{N}_2$
  Push $\mathcal{N}_2$ into Q
Return $\boldsymbol{T}$

这其中新建到 $\mathcal{N}$ 的连接和新建到 $\mathcal{N}_2$ 的连接分别对应了上面所说的第二类和第三类团.

1.3.3.2 SCT 的性质及证明

SCT 的唯一编码性 对于图和在图上构建的 SCT $\boldsymbol{T}$, 图中的每个团都能被唯一表示为 $H(T) \cup Q$. 其中 $Q \subseteq P(T)$, T 是 SCT 上一条从根到叶的路径.

我们可以看出, 每个从根到叶的路径都表示一个团, 也即 $H(T) \cup P(T)$. 其他的团可能就是这样的集合的子集. 这样的子集可能会多次出现, 但是如果我们考虑 $H(T)$ 和 $P(T)$ 的不同, 也即考虑连接的类型, 那么这种表示就是唯一的, 这也正是这个定理想要描述的内容.

证明:

考虑标签是 S 的节点 γ. 接下来我们将通过归纳 $|S|$ 证明, 每个团 $C \subseteq S$ 都可以表示为 $H(T) \cup Q$. 其中 T 是一条从 γ 到叶子的路径, $Q \subseteq P(T)$.

这里对 SCT 的归纳是从下向上的(对应 $|S|$ 从小变大), 每次添加一个父亲节点到顶上, 下面归纳情况中我们添加的其实就是前面提到的节点 γ.

1. 基本情况: S 是空的, 其他所有相关内容都是空的, 自然成立.
2. 归纳情况: 令 p 为构建 SCT 时运行到 γ 节点时选择的轴. 对于所有的团, 我们有三种情况, 分别对应了 SCT 构建时选取轴之后的三种情况.
   1. $p \in C$. 也即 γ 之下有一个标签为 $(p, \mathfrak{p})$ 的连接, 连接到 γ 的一个子节点 β. β 有标签 $N(S, p)$.
      - 存在性: 观察到 $C \setminus p$ 是 $N(S, p)$ 中的一个团. 由归纳假设, $C \setminus p$ 有一个唯一表示 $H(T) \cup Q$, 其中 T 是从 β 到叶子的路径且 $Q \subseteq P(T)$. 而且 C 没有一种这样的表示(利用从 β 到叶子的路径的), 因为 $N(S, p) \not\ni p$.

        令 T' 为包含路径 T 而且从 γ 开始的路径, 则有 $H(T') = H(T)$, $P(T') = P(T) \cup p$. 于是可以把 C 表达为 $H(T') \cup (Q \cup p)$, 其中 $Q \cup p \subseteq P(T')$.
      - 唯一性: 也即我们需要说明没有其他路径可以表示 C. 我们可以讨论上述表示中使用的路径 T
        1. T 没有经过 β: 考虑任意从 γ 开始, 但是不经过 β 的路径, 它一定经由标签为 $(v_i, \mathfrak{h})$ 的路径经过了其他子节点, 其中 v_i 是 p 的非邻居. 那么由于 $p \in C$, p 的非邻居显然不在 C 中, 所以利用这样的路径无法表示 C.
        2. T 经过 β: 如果是经过了 β 的路径, 根据归纳假设即可知其唯一.
   2. $C \subseteq N(S, p)$. 这种其实就是第一种情况的 C 去除 p, 区别就是新增节点不在 C 中. 所以表示也和第一种情况类似, 也即复用之前的表示的节点选取(从而没有加入 p), 但是使用包含 γ 的路径.
   3. C 包含一个 p 的非邻居节点.

我们先重复代码的步骤把各种符号都恢复出来: 还是令 $\{v_1, v_2, ..., v_l\} = S \setminus (N(p) \cup p)$, 令 $i = \arg\min_{j;v_j \in C} j$, 也即 v_i 是这些 v 中首个包含在 C 中的节点. 对于任意的 $1 \leq j \leq l$, 令 $N_j := N(S, v_j) \setminus \{v_1, v_2, ..., v_{j-1}\}$. 根据我们生成 SCT 的步骤, γ 下面有 l 个标签为 N_j 的孩子节点, 而且这些节点到 γ 的连接都是 $\mathfrak{h}$ 类型的, 所以对于经过 N_j 的路径 T, 有 $H(T) \ni v_j$. ($\mathfrak{h}$ 型连接的标签的节点必定选取)

- 存在性: 我们可以讨论将要在表示中使用的路径 T
  1. T 经过 $N_j, j < i$: 如果能利用 T 表示 C, 它就不能经过 $j < i$ 的 N_j, 因为 v_i 才是首个包含在 C 中的节点, v_i 之前的均不在 C 中却被选取.
  2. T 经过 $N_j, j > i$: 如果 $j > i$, 那 $N_j \not\ni v_i$ 更是没有路径能表示 C, 这是因为前面包含在 C 中的 v_i 被去掉了.
  3. T 经过 $N_j, j = i$: 由上述两种情况, 我们知道如果能利用一条路径表示 C, 那它一定经过 N_i. 注意到 $C \setminus v_i$ 是在 N_i 中的团, 由归纳假设, 有 $C \setminus v_i$ 的唯一表示 $H(T) \cup Q$, 其中 $Q \subseteq P(T)$, T 是从 N_i 开始到叶子的路径. 令 T' 为包含路径 T 而且从 γ 开始的路径, 有 $H(T') = H(T) \cup v_i$, 所以 $C = H(T') \cup Q$.
- 唯一性: 显然.

证毕.

另一方面, 每一个合法的表示显然都对应了一个团. 这其实与简单递归算法的递归树的情形类似: 因为父节点到子节点的连接标签的节点是父节点标签中的一个节点, 而子节点的标签都是他的邻居, 而且还是还是父节点标签的子集, 所以完整的路径都能表示一个团, 从里面去除一部分 p 也是如此.

1.3.3.3 利用 SCT 的唯一编码性进行团计数

根据上述关键定理, 一条从根到叶的路径 T 表示了 $2^{P(T)}$ 个不同的团. 其中大小为 $|H(T)| + i$ 的有 $\binom{P(T)}{i}$ 个, 所以我们可以轻易得出如下算法.

```
Pivoter(G):
    T ← BuildSCT(G)
    R ← [0, 0, ...]
    For T in T:
        For i ≤ |P(T)|:
            R[|H(T)| + i] ← R[|H(T)| + i] + C(P(T), i)
    Return R
```

当然, 这只是针对我的任务, 也即全局团精确计数的 Pivoter 算法. 对于边或者顶点参与的团计数, 或者列出所有团, 只需要在上述算法中稍作修改即可.

1.3.4 Pivoter 算法的复杂度分析

参见原论文.

1.3.5 Pivoter 算法的代码实现

我在网络上找到了两个 Pivoter 算法的实现:

- 论文作者给出的 C 语言实现: Bitbucket 仓库
- charunupara 给出的 Julia 实现: GitHub 仓库

另外, 我基于论文作者的 C 语言实现, 修改了一个只有全局团计数的 C++ 实现: GitHub 仓库.

不过这些实现都没有提供并行化的版本. 一些更新的算法的论文中提到了他们的算法和并行 Pivoter 算法的比较, 但是并没有提供代码.

2025-05-19

判定我变老的标准

判定我变老的标准 © 2025 by ParaN3xus is licensed under CC BY-NC-SA 4.0.

当我不加思考地贬低和否定新事物, 不再认为情况会有所改善时, 我就变老了.

2025-04-14

近乎完美的 GitLab + frp 搭建踩坑

使用 GitLab 和 frp 搭建私有服务器的经验分享

近乎完美的 GitLab + frp 搭建踩坑 © 2025 by ParaN3xus is licensed under CC BY-NC-SA 4.0.

很早之前就想自建一个 Git server, 终于在这个月早些动工了.

我的基本要求是
- 一个公网可以访问的, 全链路 HTTPS 保护的 GitLab 网站
- 可以正常使用 Git over SSH
- 有邮箱通知(尽管可能被标记为垃圾邮件)
- 服务器的 IP 不被泄露, 但是不支付额外费用购买如 Cloudflare 等平台的付费服务

为此, 我使用了
- 一台我有 root 权限的公网服务器, 用于运行 frp, wstunnel 等
- 一台我有 root 权限的内网服务器, 用于运行 GitLab 本体
- 一个二级域名
- 一个支持 SMTP 的邮箱(Gmail)

3.1 思路

基本结构如下:

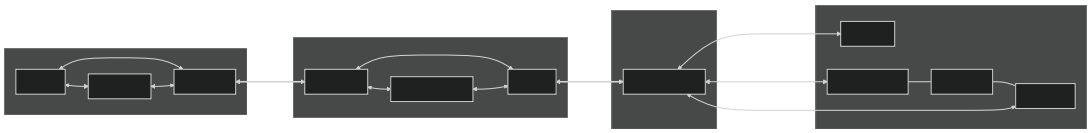

图 1　基本结构

个人认为比较重要的部分就是通过 wstunnel 代理 Git over SSH 的流量, 于是可以走 Cloudflare 达到不泄露服务器 IP 的目的.

3.2 部署过程

在开始之前, 我们先假设一些值用作示例:
- 我们给 GitLab 服务预留的域名是 gitlab.example.com
- wstunnel 服务器的 path prefix 是 ssh-tunnel
- 公网服务器的 IP 是 1.2.3.4
- 公网服务器暴露了以下端口

端口	服务
10001	frp server
80	nginx http
443	nginx https

- 公网服务器上还预留了以下端口, 但是不需要暴露

端口	服务
10002	GitLab Web
10003	GitLab SSH
10004	wstunnel server

- 内网服务器上预留了以下端口

端口	服务
801	GitLab Web
221	GitLab SSH

3.2.1 FRP Server 的安装和配置

在公网服务器上进行.

从 frp 的官方仓库下载即可.

这里提供一个简单的配置.

```
bindPort = 10001
auth.token = "some-random-password"
```

启动 frps

```
frps -c config.toml
```

3.2.2 wstunnel Server 的安装和配置

在公网服务器上进行.

从 wstunnel 的官方仓库下载即可.

可以一行启动:

```
wstunnel server ws://0.0.0.0:10004 --restrict-http-upgrade-path-prefix ssh-tunnel --
restrict-to localhost:10003
```

这里按前面所说的额外限制了 path perfix 和可以访问的端口以提高安全性.

3.2.3 GitLab 安装和配置

在内网服务器上进行.

遵循官方教程即可.

需要注意我们使用 SMTP 提供邮件服务, 所以不需要安装 postfix. 此外, 安装时首次 configurate 申请证书会失败, 这无关紧要, 我们之后使用自签名证书即可.

编辑 `/etc/gitlab/gitlab.rb` 中的这些配置项

- `external_url`: 如果你没有在安装时指定, 现在可以设置了, 我们提到过使用 `https://gitlab.example.com` 作为示例
- SMTP 相关配置: 我这里根据官方文档中的指引进行配置. 我还额外设置了 `gitlab_rails['gitlab_email_from']` 为真实的邮箱用户名.
- `letsencrypt['enable']`: 修改为 `false`. 我已经解释过.

- `nginx['listen_port']`: 如果你的服务器上还运行了其他占用 80 端口的服务, 可以把这个端口修改为其他端口, 这里使用 801 作为示例.

生成自签名证书

```
cd /etc/gitlab/ssl
sudo openssl req -x509 -nodes -days 3650 -newkey rsa:2048 -keyout
gitlab.example.com.key -out gitlab.example.com.crt
sudo chmod 600 gitlab.example.com.*
sudo chown root:root gitlab.example.com.*
```

做完这些后, 可以重新配置和重启 GitLab:

```
sudo gitlab-ctl reconfigure
sudo gitlab-ctl restart
```

3.2.4 sshd 的安装和配置

在内网服务器上进行.

直接从系统软件源安装即可.

安全起见, 我们使用一个单独的端口暴露 Git over SSH 服务, 并且只允许 git 用户登录, 这里使用 221 端口作为示例.

在 `/etc/ssh/sshd_config` 添加

```
Port 221
Match LocalPort=221
  AllowUsers git
```

重启 SSHD 服务

```
sudo systemctl restart sshd
```

3.2.5 FRP Client 的安装和配置

在内网服务器上进行.

从 frp 的官方仓库下载即可.

然后对我们用到的 nginx 和 SSH 端口做转发:

```
serverAddr = "1.2.3.4"
serverPort = 10001
auth.token = "some-random-password"

[[proxies]]
name = "gitlab-web"
type = "tcp"
localPort = 801
remotePort = 10002

[[proxies]]
name = "gitlab-ssh"
type = "tcp"
```

```
localPort = 221
remotePort = 10003
```

启动 frpc

```
frpc -c gitlab.toml
```

3.2.6 Cloudflare 的配置

在 Cloudflare 中解析 gitlab.example.com 到你的公网服务器即可.

3.2.7 nginx 的安装和配置

在公网服务器上进行.

直接从系统的软件源安装即可.

我们需要获取内网服务器上 GitLab 服务使用的自签名证书

```
sudo bash -c 'echo | openssl s_client -connect localhost:10002 -servername
gitlab.example.com 2>/dev/null | openssl x509 > /etc/nginx/ssl/gitlab_cert.pem'
```

此外, 我们还需要使用 acme.sh 申请证书, 遵从其官方指引即可, 这里不再赘述.

这里给出一个 `/etc/nginx/conf.d/gitlab.conf` 的参考配置, 具体可以根据实际情况再修改:

```
server {
    listen 443 ssl;
    server_name gitlab.example.com;

    ssl_certificate /root/.acme.sh/gitlab.example.com_ecc/gitlab.example.com.cer;
    ssl_certificate_key /root/.acme.sh/gitlab.example.com_ecc/
gitlab.example.com.key;

    location / {
        proxy_pass https://localhost:10002;

        proxy_ssl_verify off;
        proxy_ssl_trusted_certificate /etc/nginx/ssl/gitlab_cert.pem;
        proxy_ssl_verify_depth 2;

        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_read_timeout 300;
        proxy_connect_timeout 300;
        proxy_send_timeout 300;
    }

    location /ssh-tunnel {
        if ($http_upgrade = "") {
            return 404;
```

```
        }
        proxy_redirect off;
        keepalive_timeout 12000s;
        proxy_pass http://127.0.0.1:10004;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Host $host;
        proxy_set_header Connection "upgrade";
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_intercept_errors on;
        proxy_pass_request_headers on;
    }
}

server {
    listen 80;
    server_name gitlab.example.com;

    location / {
        proxy_pass http://localhost:10002;
        proxy_ssl_verify off;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_read_timeout 300;
        proxy_connect_timeout 300;
        proxy_send_timeout 300;
    }
}
```

3.2.8 客户端需要的额外配置

需要安装 wstunnel, 并且在 `~/.ssh/config` 中额外添加

```
Host gitlab.example.com
    ProxyCommand=wstunnel client wss://gitlab.example.com/ssh-tunnel --http-upgrade-
path-prefix ssh-tunnel -L stdio://127.0.0.1:10003
```

3.3 Troubleshooting

一些问题的诊断方法, 和我遇到的一些问题的解决方案.

3.3.1 我不知道 root 用户的密码

如果你忘记了在安装时指定 root 用户密码, 可以遵照 官方文档 中的方法修改密码.

3.3.2 无法访问服务

由内而外地诊断哪里出了问题. 比如先在内网服务器上 `curl http://localhost:801`, 测试通过再在公网服务器上测试 `localhost:10002`, `localhost:80`.

Git over SSH 的问题也可以以同样的方式诊断.

3.3.3 GitLab 无法保存配置, 错误代码 500

看上去是一些 token 错误引起的问题, 总体的解决方案就是删掉这些 token. 我尝试了若干方法, 已经不太清楚哪一步起了作用, 这里是我当时查阅过的内容:

- https://gitlab.com/gitlab-org/gitlab/-/issues/419923
- https://gitlab.com/gitlab-org/gitlab/-/issues/334862
- https://gitlab.com/gitlab-org/gitlab/-/issues/301170
- https://forum.gitlab.com/t/500-error-access-admin-runners-not-a-migration/100875

3.3.4 注册邮件无法正常发送

可以按照 GitLab 官方文档 中的步骤进行诊断.

如果最后发现 `Notify.test_email` 无法正常发信, 可以使用第三方工具比如 swaks 按相同的配置发信看看有无更详细的错误信息.

2025-02-28

宝宝的强化学习

简单的强化学习入门, 包括马尔可夫决策过程, Q-Learning 和 DQN 算法的介绍和实现.

宝宝的强化学习 © 2025 by ParaN3xus is licensed under CC BY-NC-SA 4.0.

你可能或多或少听说过(或者十分了解)“监督学习”, 也就是给出很多自变量-因变量对, 然后通过某种方式拟合那个函数.

但是监督学习并不能解决所有问题, 尤其是当我们没有那么多自变量-因变量对时候. 如果能把我们的问题转换成模型做出一些“动作”, 从而和“环境”交互, 而且我们能较轻易地得知交互结果的好坏(无论是最终地还是暂时地), 我们能不能从这些结果的好坏中获得经验, 从而优化模型的行为呢?

4.1 环境? 动作? 结果?

我们暂时承认这个想法有可能是可行的, 但是为了搞清楚他是不是真的可行, 我们需要做一些严肃的推理和实验.

因此, 我们要先**形式化**这个问题.

我们用一个状态量 $s \in S$ 来描述环境, 模型采取的行动 $a \in A$ 会让 s **改变**, 而且每次改变, 模型都会得到 $r(s, a)$ 的奖励(或者惩罚). 具体是怎么**改变**呢, 比较好(通用)的描述应该是产生一个下一状态的概率分布 $P(s' \mid s, a)$.

由于这显然是一个时序的过程, 所以我们定义时间步 $t < T$, 另外有在第 t 步时的状态为 S_t, 模型行动为 A_t, 获得的奖励为 $R_t = r(S_t, A_t)$.

为了控制模型对近期和远期利益的倾向, 我们引入一个折扣因子 γ, 并且定义从时刻 t 开始直到结束, 模型获得的总回报为

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{1}$$

还有最重要的, 我们的模型, 或者说“策略”: π. 我们将其定义为给定状态下采取各种行动的概率的分布, 也即 $\pi(\cdot \mid s)$.

实际上, 我们定义的 $\langle S, A, P, r, \gamma \rangle$ 就是一个**马尔可夫决策过程**.

4.2 更多回报, 但是回报有多少?

我们希望得到更好的策略. 具体来说, 就是让该策略能够得到更大的总回报. 这种倾向反映在策略中, 也就是让一些能带来更大总回报的行动 a 的概率更高.

而在状态 s 上执行动作 a 时, 遵循策略 π 的总回报期望(**动作价值函数**)是

$$
\begin{aligned}
Q^{\pi}(s,a) &= \mathbb{E}(G_t \mid s_t = s, a_t = a) \\
&= \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k} \,\middle|\, s_t = s, a_t = a \right) \\
&= \mathbb{E}\left(R_t + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, s_t = s, a_t = a \right) \\
&= r(s,a) + \gamma \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, s_t = s, a_t = a \right)
\end{aligned} \tag{2}
$$

这里 s_{t+1} 是一个分布, 如果能得知在某个状态上执行策略 π 的总回报期望(**状态价值函数**), 我们还能继续分解上式末尾的 $\mathbb{E}(\cdot)$, 于是我们定义这个值

$$
V^{\pi}(s) = \sum_{a \in A} \pi(a \mid s) Q^{\pi}(s,a) \tag{3}
$$

正如刚刚说的, 我们可以继续变形 Q:

$$
\begin{aligned}
Q^{\pi}(s,a) &= r(s,a) + \gamma \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, s_t = s, a_t = a \right) \\
&= r(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^{\pi}(s')
\end{aligned} \tag{4}
$$

现在看看我们得到了什么

$$
Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^{\pi}(s') \tag{5}
$$

$$
V^{\pi}(s) = \sum_{a \in A} \pi(a \mid s) Q^{\pi}(s,a) \tag{6}
$$

4.3 那么最优策略呢?

在我们推导了太多和一般策略有关的期望之后, 我们终于准备好研究一般策略的特例: **最优策略**.

事不宜迟, 我们定义一个策略 π^*, 使得 $\forall \pi, \forall s \in S, V^{\pi^*}(s) \geq V^{\pi}(s)$. 这是非常直观的最优策略, 因为无论在什么状态下, 其表现(回报期望)都比任意策略更好, 或至少一样.

根据该定义, 我们就有

$$
V^*(s) = \max_{\pi} V^{\pi}(s) \tag{7}
$$

和

$$
Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{8}
$$

一个好消息是, 既然最优策略是策略, 那他就符合我们上面推导出的一些结论, 比如 式 5, 于是我们有

$$
Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^*(s') \tag{9}
$$

另一个好消息是, 既然最优策略是最优的, 他就会在每个状态上都采取 Q 最大的 a. 那么除非若干个行动有相同的 Q, $\pi^*(\cdot \mid s)$ 将会是 one-hot 的(只有一个行动的概率为 1, 其他都为 0), 否则其 V, 根据 式 6, 将并非最大. 无论是多个相同 Q 的行动分享 1 还是 one-hot, 都有

$$V^*(s) = \max_{a \in A} Q^*(s, a) \tag{10}$$

这其实是 式 6 的简化版本.

式 9 和 式 10 看起来是两个互相耦合的函数, 不过既然如此, 我们也可以把它们互相代入, 得到递归的形式.

式 9 $\to$ 式 10

$$V^*(s) = \max_{a \in A} \left(r(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^*(s') \right) \tag{11}$$

式 10 $\to$ 式 9

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) \max_{a' \in A} Q^*(s', a') \tag{12}$$

这两个式子就是**贝尔曼最优方程**.

4.4 如何求出它?

策略是抽象的, 听上去似乎不像是什么可以求解的对象, 尤其是当我们并没有讨论具体问题的时候. 但是通过上面的这些推演, 我们已经把抽象的策略变成了具体的数学对象 Q 和 V: 只要得到这两个函数中的一个(因为他们可以互相推导), 我们就事实上得到了最优策略, 因为我们可以选择 $\arg\max_a Q(s, a')$ 来让回报最大化.

下面我们不加证明地给出 Q-Learning 的迭代公式.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[R_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big] \tag{13}$$

当 Q-Learning 算法工作时, 我们先从一个随机初始化的 Q 和状态 s_t 开始, 将 Q 当作 Q^* 执行策略, 也即执行 $a_t = \arg\max_a Q(s_t, a_t)$. 每次得到环境的反馈 R_t, s_{t+1} 后, 就进行一次迭代, 然后重复这个过程.

虽然我并不打算细讲这个迭代公式的收敛性的具体证明过程, 但是我们还是有必要从直观上理解这个公式. 我们主要关注增量部分, 它包含

- α: 这是一个可调参数, 用于控制学习率, 这很好理解
- $R_t + \gamma \max_a Q(S_{t+1}, a)$: 利用环境给出的信息 R_t 展开 Q 的一项, 这是我们给出的新的 Q 的估计. 由于它正确地利用了环境给出的信息, 所以直觉上应该比原始的 Q 更接近最优
- $-Q(s_t, a_t)$: 减去原本的 Q 的估计, 得到增量部分, 这很好理解

但是这还不够, 事实上 Q-Learning 在选择动作时并没有完全按照估计的 Q 执行——算法执行初期我们希望得到更多环境的信息, 所以我们需要一些随机探索.

所以实际上 Q-Learning 采用的是 ε-贪心算法, 也就是有 ε 的概率随机选择 $a \in A$ 执行, 剩余的 $1 - \varepsilon$ 则贪心地选择 $\arg\max_a Q(s_t, a_t)$, 然后我们可以令 ε 随时间衰减, 就能达到我们"在算法运行初期增加随机探索"的目的了.

然而, Q-Learning 有个巨大的缺点: 他只能处理 S 和 A 都是有限集合的情形, 因为迭代公式只是在更新单个自变量的值. 这真是太坏了, 因为现实中的很多问题都是连续的, S 和 A 都是巨大的实向量空间, 这种情况下我们显然不能更新单个自变量对应的函数值来得到 Q, 我们要怎么办?

4.5 函数拟合? 有了

据我们所知, 神经网络很擅长这种拟合函数的工作. 但是要在这种问题上采用神经网络, 我们还需要得知网络的更新方式: 我们需要损失值.

损失值基本上就是模型输出和正确结果(至少是"更正确")之间的差异, 带着这种观点重新审视式 13, 我们会发现答案就在其增量部分. 只要简单地套上一个 MSE, 我们就找到了损失值

$$L = \mathrm{MSE}\Big(R_t + \gamma \max_a Q(s_{t+1}, a), Q(s_t, a_t)\Big) \tag{14}$$

我们已经很接近 **DQN**(Deep Q Network) 算法了, 还剩下一点实践中的考量

4.5.1 不要浪费样本

神经网络当然不能像 Q-Learning 那样采样一次就利用这次采样的数据迭代一次: 这样得到的数据并不"同分布", 于是神经网络只能学习到最近的数据; 每个样本只使用了一次, 效率太低.

所以, 我们会先连续地在环境中采样, 并把采样得到的 (s_t, a_t, R_t, s_{t+1}) 保存到一个**缓冲区**中, 等到缓冲区中有一定数量的样本后, 再进行训练.

4.5.2 "参考答案"也有网络的输出

我们这个神经网络的训练和监督学习的训练略有不同: MSE 的两项都包含神经网络本身的输出, 更新网络的时候目标也在改变, 这会导致训练不稳定.

所以我们干脆使用两套网络, 一套**目标网络**暂不更新, 用于计算 $\max_a Q(s_{t+1}, a)$, 另一套正常更新, 用于计算 $Q(s_t, a_t)$. 只有每隔 C 步, 才把目标网络和正常更新的网络进行同步, 从而保证训练的稳定性.

4.6 我想试试看

我已经跃跃欲试了, 有没有什么问题简单又适合 DQN 的让我做一做?

有的兄弟, 有的. 下面我们来看一个经典问题: 车杆问题(Cart-Pole Problem).

车杆问题(如 图 2)是一个经典控制问题, 其基本环境由一个可在水平轨道上左右移动的小车和一根铰接在小车上的直杆组成. 杆的初始状态略有倾斜, 因此会因重力而自然倾倒. 我们的目标是通过控制小车的左右移动, 使杆保持竖直平衡状态尽可能长的时间, 要求杆与竖直方向的夹角不超过特定阈值, 同时小车不能超出轨道边界.

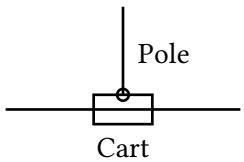

图 2　车杆问题示意图

4.6.1 定义环境

好消息是 Python 包 `gymnasium` 已经为我们实现了这个环境的代码, 我们可以直接调用.

```
from gymnasium.envs.classic_control import CartPoleEnv
from gymnasium.wrappers.common import TimeLimit
```

```
def get_env(render: False, max_step=-1):
    raw_env = CartPoleEnv(render_mode="human" if render else None)
    if max_step > 0:
        return TimeLimit(raw_env, max_episode_steps=max_step)
    return raw_env
```

这里不使用 `gym.make("CartPole-v1")` 的原因是其限制了最长步数为 500, 这样的步数仍然过短, 模型可能陷入局部最优解, 比如任小车以较慢的速度滑出有效区域.

这个环境的 S 和 A 如下表所示.

状态	取值区间
车的位置	$[-4.8, 4.8]$
车的速度	$\mathbb{R}$
杆的角度	$[-0.418, 0.418]$
杆末端的角速度	$\mathbb{R}$

表 2　车杆问题状态空间

动作	值
向左推车	0
向右推车	1

表 3　车杆问题动作空间

只要坚持一帧, 就能获得分数为 1 的奖励.

4.6.2 设计 Q 网络

根据环境的 S 和 A, 我们需要设计一个接收一个四维向量, 并输出一个二维向量的网络. 我这里给出一个例子.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNet(nn.Module):
    def __init__(self, state_size=4, action_size=2):
        super(QNet, self).__init__()

        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, mode='fan_in',
nonlinearity='relu')
                nn.init.constant_(m.bias, 0)
```

```
    def forward(self, state):
        if state.dim() == 1:
            state = state.unsqueeze(0)

        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        q_values = self.fc3(x)

        return q_values
```

这是一个具有两个隐藏层的网络, 并使用 ReLU 作为激活函数, 还使用了 He 初始化, 优化了初期的训练.

4.6.3 样本缓冲区

我们用 `collections.deque` 做一个简单的样本缓冲区, 可以向里面存入样本, 然后随机地取出.

```
import collections
import random
import numpy as np

class SampleBuffer:
    def __init__(self, max_size):
        self.buffer = collections.deque(maxlen=max_size)

    def add(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        samples = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*samples)
        return np.array(state), action, reward, np.array(next_state), done

    def size(self):
        return len(self.buffer)
```

4.6.4 DQN 算法

正如我们前面所说的, 我们实现 DQN 算法的更新算法和ε-贪心策略. 有一点不同的是, 我们的环境会因为某些原因终止, 比如杆的角度或者小车位置超出范围, 到达最大时间等. 当环境终止时, 要把 式 14 MSE 中第一项(也即下面代码的 `q_targets`) 中对未来的估计部分变为 0, 因为环境已经终止, 未来不会有任何回报了.

```
class DQN:
    def __init__(self, state_dim, action_dim, learning_rate, gamma, epsilon,
target_update_freq, device):
        self.action_dim = action_dim

        self.q_net = QNet(state_dim, action_dim).to(device)
        self.target_q_net = QNet(state_dim, action_dim).to(device)

        self.optimizer = torch.optim.Adam(
            self.q_net.parameters(), lr=learning_rate)
        self.gamma = gamma
        self.epsilon = epsilon
```

```
        self.target_update = target_update_freq
        self.update_count = 0
        self.device = device

    def take_action(self, state):
        # epsilon-greedy
        if np.random.random() < self.epsilon:
            action = np.random.randint(self.action_dim)
        else:
            state = torch.tensor([state]).to(self.device)
            action = self.q_net(state).argmax().item()
        return action

    def update(self, transition_dict):
        states = torch.tensor(transition_dict['states']).to(self.device)
        actions = torch.tensor(
            transition_dict['actions']).view(-1, 1).to(self.device)
        rewards = torch.tensor(
            transition_dict['rewards']).view(-1, 1).to(self.device)
        next_states = torch.tensor(
            transition_dict['next_states']).to(self.device)
        dones = torch.tensor(
            transition_dict['dones']).view(-1, 1).to(self.device)

        # Q(s_t, a_t)
        q_values = self.q_net(states).gather(1, actions)

        # max_a Q(s_(t + 1), a)
        max_next_q_values = self.target_q_net(
            next_states).max(1)[0].view(-1, 1)

        # r(s, t) + gamma max_a Q(s_(t + 1), a), mul (1-done) for obvious reason
        q_targets = rewards + self.gamma * max_next_q_values * (1 - dones)

        loss = F.mse_loss(q_targets, q_values)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # update target network
        if self.update_count % self.target_update == 0:
            self.target_q_net.load_state_dict(self.q_net.state_dict())
        self.update_count += 1
```

4.6.5 开始训练

设置参数, 初始化模型, 然后根据我们前面所述的策略启动训练.

```
import tqdm.notebook as tqdm

device = torch.device(
    "cuda") if torch.cuda.is_available() else torch.device("cpu")

lr = 2e-3
num_episodes = 500
```

```
gamma = 0.98
epsilon = 0.01
target_update = 10
buffer_size = 10000
min_buffer_size = 500
batch_size = 64

env = get_env(render=False, max_step=2000)
replay_buffer = SampleBuffer(buffer_size)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
agent = DQN(state_dim, action_dim, lr, gamma, epsilon,
            target_update, device)

return_list = []
for i in range(10):
    with tqdm.tqdm(range(int(num_episodes / 10)), desc='Iteration %d' % i) as pbar:
        for i_episode in pbar:
            episode_return = 0
            state, _ = env.reset()
            done = False
            while not done:
                action = agent.take_action(state)
                next_state, reward, terminated, truncated, _ = env.step(action)
                done = terminated or truncated
                replay_buffer.add(state, action, reward, next_state, 1 if done else
0)
                state = next_state
                episode_return += reward

                # train after there are enough samples
                if replay_buffer.size() > min_buffer_size:
                    b_s, b_a, b_r, b_ns, b_d = replay_buffer.sample(batch_size)
                    transition_dict = {
                        'states': b_s,
                        'actions': b_a,
                        'next_states': b_ns,
                        'rewards': b_r,
                        'dones': b_d
                    }
                    agent.update(transition_dict)
            return_list.append(episode_return)
            if (i_episode + 1) % 10 == 0:
                pbar.set_postfix({
                    'episode':
                    '%d' % (num_episodes / 10 * i + i_episode + 1),
                    'return':
                    '%.3f' % np.mean(return_list[-10:])
                })
```

4.6.6 观察结果

新建一个有可视化界面的环境, 然后看看模型的表现吧!

```
env = get_env(render=True)
```

```
state, _ = env.reset()
done = False
while not done:
    action = agent.take_action(state)
    env.render()
    next_state, reward, terminated, truncated, _ = env.step(action)
    done = terminated or truncated
    state = next_state
```

不出意外的话, 你的模型应该能很好地控制住杆, 直到永远.

如果不幸没有, 请再仔细检查代码有没有错误. 此外由于我并没有设置任何 `seed`, 所以这种情况也是完全有可能发生的, 可以再尝试几次, 或者尝试修改超参数.

无论如何, it works on my machine.

4.7 好像还缺点什么

我们批评过 Q-Learning 只能学习离散状态空间中的 Q 函数, 但现在我们的 DQN 也不能给出连续的动作.

是的, 所以研究者们还提出了基于"策略梯度"的算法(DQN 是基于值函数的), 比如 REINFORCE, 这种算法可以预测连续的动作. 除此之外, 还有混合两种思想的算法, 比如 Actor-Critic, PPO, DDPG 等. 但是这些算法都不在本文的计划范围之内了.

至此, 我们已经完成了对强化学习基础概念和两种入门算法的介绍. 从最初的马尔可夫决策过程, 到价值函数, 策略函数的概念, 再到 Q-Learning 和 DQN 算法的实现, 我们循序渐进地窥见了强化学习的一点思想.

不管怎样, 强化学习是一个广阔而深刻的领域, 本文也将仅仅止步与其思想和入门算法的介绍. 上面提到的其他算法并不在本文的计划内容范围内. 若有兴趣, 可以自行了解.

希望你有所收获.

2025-02-05

florr.io 中的合成与概率

在 WSL 中创建一个 alias, 直接使用 Windows 文件资源管理器打开目录或文件.

florr.io 中的合成与概率 © 2025 by ParaN3xus is licensed under CC BY-NC-SA 4.0.

florr.io 是一款~~开放世界冒险游戏~~, 玩家可以在自己控制的花朵(flower)上装备各种花瓣(petal)来与地图中昆虫或者其他物体战斗.

花瓣有不同种类, 相同的种类也有等级高低之分, 高等级的花瓣可以通过击败高等级的昆虫, 并拾取其掉落物获得, 也可以通过五个次一级的同种花瓣合成获得.

5.1 花瓣合成的机制

花瓣合成的具体机制如下:

1. 一次合成是否成功服从伯努利分布, 对于大多数花瓣来说, 各等级合成的成功概率为

合成等级	成功率
普通合成罕见	64%
罕见合成稀有	32%
稀有合成史诗	16%
史诗合成传奇	8%
传奇合成神话	4%
神话合成究极	2%
究极合成超级	1%

表 4　花瓣合成成功率表

2. 如果合成失败, 将会随机销毁一个或数个花瓣, 销毁的花瓣数服从 $U(1,4)$.

为了后续讨论的方便, 这里我们假设每次合成都是独立的, 也即不存在“保底”等规则.

5.2 我需要多少次级花瓣

有些时候, 我们想要知道需要多少次级的花瓣才能有较大概率(比如 95%)获得至少一个高级花瓣.

这里我们设一次合成成功率为 p, 为了有 k 的概率获得至少一个高级花瓣, 我们有

$$1-k=(1-p)^{m_k}$$

$$\Rightarrow m_k=\frac{\ln(1-k)}{\ln(1-p)}$$

其中 m_k 是预期的合成次数.

这 m_k 次中包含了 m_k-1 次合成失败和 1 次合成成功, 对于每次合成失败, 预期的损耗为 $\frac{5}{2}$, 所以

$$
\begin{aligned}
n_k &= \frac{5}{2}(m_k - 1) + 5 \\
&= \frac{5}{2}\left(\frac{\ln(1-k)}{\ln(1-p)} + 1\right)
\end{aligned}
$$

其中 n_k 是需要的次级花瓣数.

根据此公式, 我们得到如下表格

合成等级	成功率	$n_{0.95}$	$n_{0.99}$
普通合成罕见	64%	9.83	13.77
罕见合成稀有	32%	21.92	32.35
稀有合成史诗	16%	45.45	68.53
史诗合成传奇	8%	92.32	140.58
传奇合成神话	4%	185.96	284.53
神话合成究极	2%	373.21	572.37
究极合成超级	1%	747.68	1148.03

表 5　花瓣合成所需次级花瓣数表