图 Graph

1 定义与实现

图 G = (V, E) ，其中 V 是顶点集合，E 是边集合

1.1 常用术语

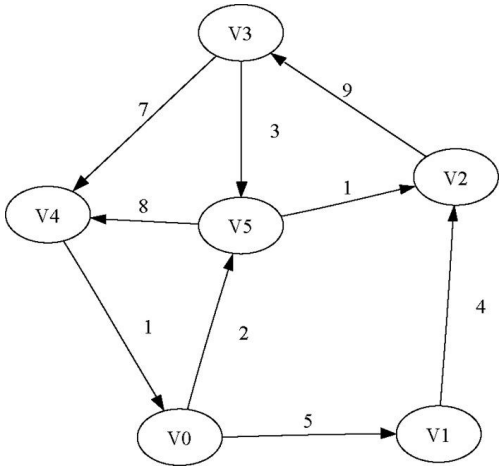
- 顶点 (vertex)
    - 度数 / 入度、出度
- 边 (edge)
- 权重 (weight)
- 路径 (path)
- 环 (ring)
    - 无环图
        - 有向无环图 (DAG, directed acyclic graph)

1.2 实现

1.2.1 方法

- `Graph()`新建一个空图。
- `addVertex(vert)`向图中添加一个顶点实例。
- `addEdge(fromVert, toVert)`向图中添加一条有向边，用于连接顶点`fromVert`和`toVert`。
- `addEdge(fromVert, toVert, weight)`向图中添加一条带权重`weight`的有向边，用于连接顶点`fromVert`和`toVert`。
- `getVertex(vertKey)`在图中找到名为`vertKey`的顶点。
- `getVertices()`以列表形式返回图中所有顶点。
- `in`通过`vertex in graph`这样的语句，在顶点存在时返回`True`，否则返回`False`。

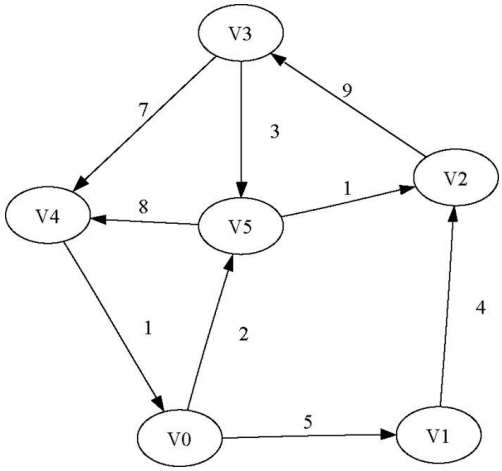
1.2.2 用邻接矩阵实现

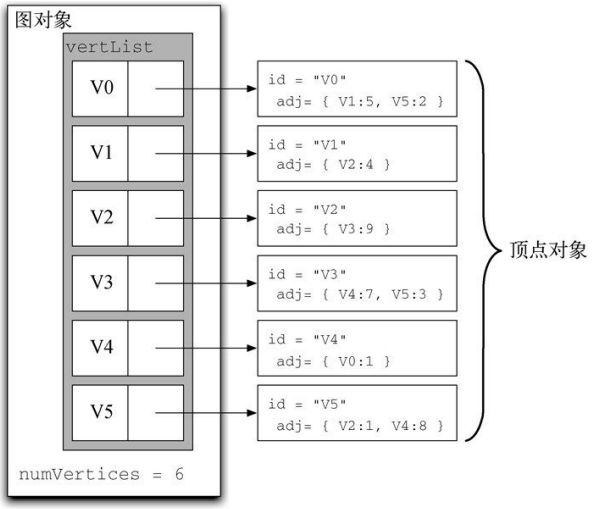


	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

问题：矩阵稀疏，不高效

1.2.3 用邻接表实现





为图对象的所有顶点保存一个主列表，同时为每一个顶点都维护一个列表（图中用了字典）记录与它相连的顶点。

具体实现见 这里 （ `Vertex` 类和 `Graph` 类）

2 宽度优先搜索 BFS

2.1 词梯问题

3 深度优先搜索 DFS

3.1 骑士周游问题

3.2 通用深度优先搜索

3.3 拓扑排序

根据有向无环图生成一个包含所有顶点的线性序列，使得如果图 G 中有一条边为 (v, w)，那么顶点 v 排在顶点 w 之前。在很多应用中，有向无环图被用于表明**事件优先级**。

- 算法

    1. 对图 G 调用 dfs ，计算每一个顶点的结束时间
    2. 基于结束时间，将顶点按照递减顺序存储在列表中
    3. 将有序列表作为拓扑排序的结果返回

4 强连通分量 SCC

在有向图 G 中，如果两点互相可达，则称这两个点*强连通*，如果 G 中任意两点互相可达，则称 G 是强连通图。非强连通有向图的极大强连通子图，称为*强连通分量* (SCC, Strongly Connected Component)。

把强连通分量中的所有顶点组合成单个顶点，**从而将图简化**。

4.1 Tarjan 算法

- 算法：用栈跟踪已访问过的节点，`disc[v]` 记录顶点 v 的发现时间，`low[v]` 记录 v 或 v 的**子树**能够回溯到的最早的栈中节点。

  对图 G 调用 `dfs` ，对于每个未访问的顶点 v：

  1. 将 `disc[v]` 和 `low[v]` 记录为当前 dfs 时间；增加 dfs 时间的计数
  2. 将 v 推入栈中
  3. 遍历顶点的邻居 neighbor：
     (a) 若邻居未访问：
        i. 递归地对其 dfs 遍历
        ii. 更新 `low[v]` ： `low[v] = min(low[v], low[neighbor])`
     (b) 若邻居在栈中：
        i. 更新 `low[v]` ： `low[v] = min(low[v], disc[neighbor])`

        > 注意为什么是 `disc[neighbor]` 而不是 `low` ！（仅仅求 SCCs 时好像无所谓，但在利用这一算法的其它情况下好像会出问题）
  4. 遍历完所有邻居后，检查当前顶点是否是一个强连通分量的根：

     若 `disc[v] == low[v]` ，则 v 是一个强连通分量的根。从栈中弹出顶点直到当前顶点，形成并存储一个强连通分量

- 实现：详见 这里
- 时间复杂度：$O(V+E)$

4.2 Kosaraju 算法

- 算法

  1. 对图 G 调用 `dfs` ，计算每一个顶点的结束时间
  2. 计算 G 的转置 G'
  3. 对图 G' 调用 `dfs` ，按照结束时间的递减顺序访问顶点
  4. 第3步得到的深度优先森林中的每一棵树都是一个强连通分量

- 解释 (by ChatGPT)

  - The first DFS traversal gives us the finishing times of vertices, which essentially **provides a reverse topological order** of the graph.
  - In the second DFS traversal, we explore the transpose graph G' **starting from the 'sink' vertices** (those with the highest finishing times), which guarantees that we cover all vertices within each strongly connected component before moving to the next sink vertex.
  - The algorithm takes advantage of the fact that in the transpose graph G', the sink vertices of each strongly connected component become the **roots** of the trees in the DFS forest, and each exploration from a sink vertex covers the **entire** component.

- 实现：详见 这里

- 时间复杂度：$O(V+E)$ ，但因为完整 dfs 了两次，应该说是 asymptotically optimal 。在实际的测试中，Tarjan 算法的运行效率比 Kosaraju 算法高 30% 左右

5 最短路径问题

5.1 Dijkstra 算法

提供从一个顶点到其他所有顶点的最短路径

- 算法：利用优先级队列维护循环顺序。将起始顶点的距离初始化为 0 ，其他所有顶点的距离初始化为无穷大。将所有顶点放入一个优先级队列（最小堆）中。

  1. 从队列中 pop 出最小元 v（第一次即为起始顶点）
  2. 更新与 v 相连的顶点的距离信息（取 min），同时维护队列
  3. 重复 1-2 ，直到队列为空

- 实现：详见 这里

- 时间复杂度：$O((V+E)\log V)$

6 最小生成树

A **spanning tree** is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.

称带权连通无向图 G 的所有生成树中权值和最小的生成树为 G 的*最小生成树* (Minimum Spanning Tree，MST)。

6.1 Prim 算法

从某一个**顶点**开始构建生成树，每次将代价最小的新顶点纳入生成树，直到全部纳入为止。

- 算法：利用*优先级队列*维护待加入顶点的顺序。初始化空的 set `mst` 记录最小生成树，空的 priority queue `pq` 记录可加入的边。

  1. 任意选取起始顶点，将其相连的边加入 pq 中
  2. 从 pq 中 pop 出具有最小权重的边 e
  3. 若 e 连接的顶点 v 不在 mst 中，将其加入，并将 v 相连的边加入 pq 中
  4. 重复 2-3 步，直到 pq 为空

- 实现：详见 这里

- 时间复杂度：$O(V^2)$ ，适用于稠密图

6.2 Kruskal 算法

每次选择一条权值最小的**边**，将这条边的两头连通(不选已经连通的)，直到所有节点连通。

- 算法：利用*并查集* (Disjointset) 记录连通情况。初始化空的 set `mst` 记录最小生成树，将所有边按权值从小到大排序。将所有顶点视为孤立的连通集。

  1. 从小到大依次取出边 e
  2. 若加入 e 不会在 mst 中形成环（体现为 e 两端的顶点不位于同一个并查集内），将其加入 mst 中，并将其两端顶点所在的集合合并为一个
  3. 重复 1-2 步，直到所有边都已处理

- 实现：详见 这里

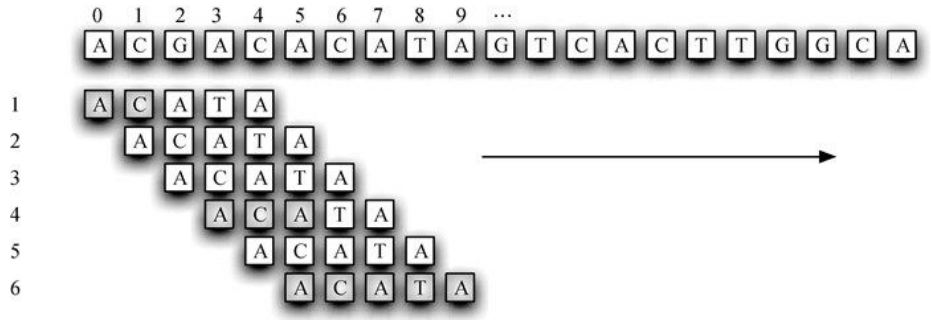
- 时间复杂度：$O(E \log E)$ ，适用于稀疏图

7 模式匹配

在长字符串中寻找模式。这种模式常被称作子串。

- 以下均以 DNA 串的模式匹配为例

7.1 简单比较



- 时间复杂度 $O(mn)$
- 如果对模式有一定的了解，可先对其做一些预处理，就可以创建时间复杂度为 $O(n)$ 的模式匹配器

7.2 确定有限状态自动机 DFA

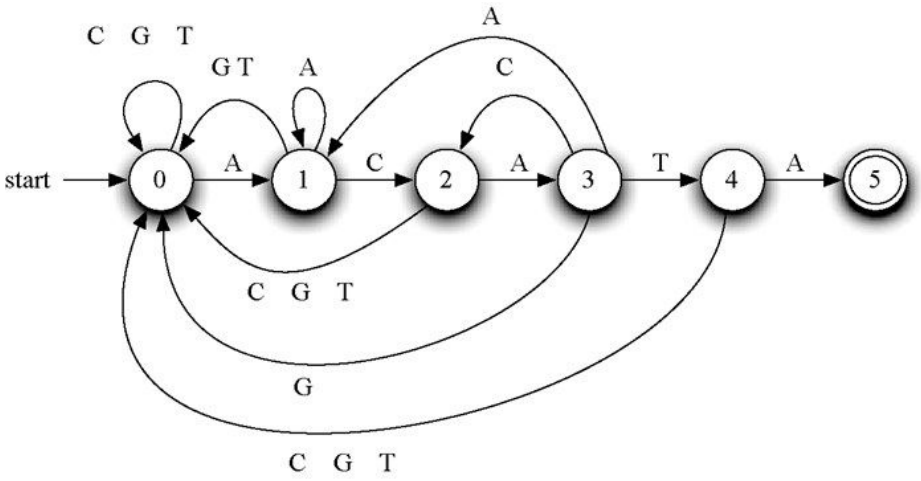
- Deterministic Finite Automation

在 DFA 图中，每个顶点是一个状态，用于记录匹配成功的模式数；每一条边代表处理文本中的一个字母后发生的转变。

- DFA 图用状态 0 表示初始状态，用两个同心圆表示最终状态

7.2.1 原理

记录当前状态，并在一开始时将其设为 0 。读入下一个字母，根据这个字母，相应地转变为下一个状态，并将它作为新的当前状态。由定义可知，对于每个字母，每个状态有且只有一种转变。这意味着对于基因字母表，每个状态可能有 4 种转变。如下图所示，我们在某些边上标出了多个字母，表示到同一个状态的多种转变。重复上述做法，直到终止（进入最终状态或穷尽字母）。



7.2.2 分析

- 已有 DFA 图，时间复杂度 $O(n)$
- 但还需考虑构建 DFA 图的预处理步骤
  - 有很多知名算法可以根据模式生成 DFA 图。不幸的是，它们都很复杂。

7.3 KMP 模式匹配算法

- Named after D. E. **K**nuth，J. H. **M**orris & V. R. **P**ratt.

7.3.1 教材上的分析与解释

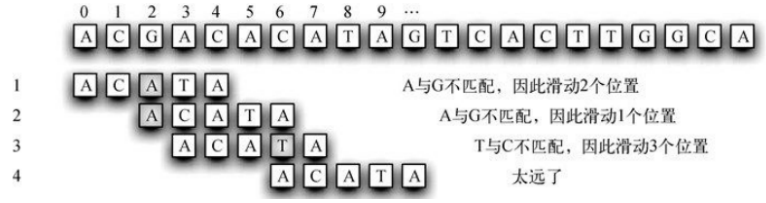
> 8.6.4 使用图：KMP
>
> 8.6.2节中的模式匹配器将文本中的每个可能匹配成功的子串都与模式比较。这样做往往是在浪费时间，因为匹配的实际起点远在之后。一种改善措施是，如果不匹配，就以多于一个字母的幅度滑动模式。图8-24展示了这种策略，将模式滑到前一次发生不匹配的位置。
>
> 
>

图8-24　滑动幅度更大的模式匹配器

第1步，我们发现前两个位置是匹配的。不匹配的是第3个字母（图中带阴影的字母），将整个模式滑过来，下一次尝试从这一点开始。第2步，一上来就不匹配，没有其他选择，只能滑动到下一个位置。此时，我们发现前3个字母是匹配的。但有个问题：不匹配时，算法根据策略把模式滑到某个位置；但这样做就滑过头了，也就是漏掉了模式在文本串中真正的起点（位置5）。

这个方案失败的原因在于，没有利用前一次不匹配时模式和文本的内容信息。在第2步中，文本串的最后2个字母（位置5和位置6）实际上是和模式的前2个字母匹配的。我们称这种情况为模式的两字母前缀与文本串的两字母后缀匹配。这条信息很有价值。如果记录下前缀和后缀的重叠情况，就可以直接将模式滑动到正确位置。

基于上述思路，可以构建名为KMP（Knuth-Morris-Pratt）的模式匹配器，它以提出这一算法的3位计算机科学家的姓氏命名。KMP算法的思想就是构建图，在字母不匹配时可以提供关于“滑动”距离的必要信息。KMP图也由状态（顶点）和转变（边）构成。但不同于DFA图，KMP图的每个顶点只有2条向外的边。

图8-25是示例模式的KMP图，其中有两个特殊的状态。初始状态（标有get的顶点）负责从输入文本中读入下一个字母。随后的转变（标有星号的边）是肯定发生的。注意，一开始从文本读入前两个字母，然后立即转到下一个状态（状态1）。最终状态（状态6，标有F的顶点）表示匹配成功，它对于图来说是终点。

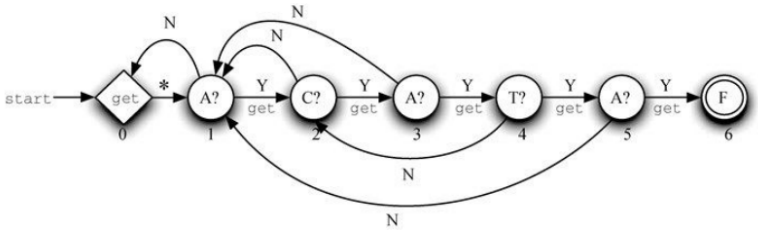


图8-25　KMP图示例

其他顶点负责比较模式中的每个字母与文本中当前的字母。例如，标有C?的顶点检查文本中当前的字母是否为C。如果是，就选择标有Y的边（Y代表yes，说明匹配成功），同时读入下一个字母。无论是否匹配成功，都会从文本中读入下一个字母。

标有N的边表示不匹配。前面解释过，遇到这种情况时，要知道滑动多少个位置。本质上，我们是要记录文本中当前的字母，并且往回移动到模式中的前一个点。为了计算，我们采用一个简单的算法（如代码清单8-29所示），比较模式与其自身，找出前缀和后缀的重叠部分。由重叠部分的长度可知要往后挪多远。注意，使用不匹配链接时，不处理新的字母。

（我好像没看懂）

7.3.2　failure_table

关键是依据模式构建 `failure_table` (就是上面所说的图)，以减少没必要的比较。

- `failure_table[i]` 记录模式中以下标 i 结尾的、与模式前缀匹配的最长字串
  - 例：Pattern: "AAABAAA"
    - `failure_table[0] = 0` (since the first character "A" has no proper suffix that is also a prefix)
    - `failure_table[1] = 1` (the longest proper suffix that is also a prefix for the substring "AA" is "A" of length 1)
    - `failure_table[2] = 2` (the longest proper suffix that is also a prefix for the substring "AAA" is "AA" of length 2)
    - `failure_table[3] = 0` (the longest proper suffix that is also a prefix for the substring "AAAB" is empty, hence 0)

- `failure_table[4] = 1` (the longest proper suffix that is also a prefix for the substring "AAABA" is "A" of length 1)
- `failure_table[5] = 2` (the longest proper suffix that is also a prefix for the substring "AAABAA" is "AA" of length 2)
- `failure_table[6] = 3` (the longest proper suffix that is also a prefix for the substring "AAABAAA" is "AAA" of length 3)

  - 实现

```
arr = [None] * len(w)
def failure_table(w):
    m = len(w)
    j = 0
    i = 1
    global arr
    arr[0] = 0

    while i < m:
        if w[j] == w[i]:    #
            j += 1
            arr[i] = j
            i += 1
        elif j == 0:        #
            arr[i] = 0
            i += 1
        else:               #
            j = arr[j - 1]
```

- KMP 实现

```
def kmp_search(w, s):
    failure_table(w)
    j, i = 0, 0
    m = len(w)
    n = len(s)

    while j < m and i < n:
        if w[j] == s[i] and j == m - 1:  #
            return True
        elif w[j] == s[i]:               #
            j += 1
            i += 1
        else:                            #
            if j != 0:
                j = arr[j-1]
            else:
                i += 1
```

```
        return False
```

7.3.3 总实现

见 这里 （ i 和 j 和这里是反的哈）

7.3.4 分析

- 时间复杂度 $O(n+m)$
- KMP 图易于构建