编译系统与正则表达式的图论建模

问题求解（三）第 1 周 Open Topic

黄文睿　221180115

南京大学

主要内容

1.1 编译系统的概念

情景提要

- 当一个项目只有一个源文件，编译是容易的；
- 当一个项目有多个源文件，编译会变得复杂：编译有了依赖关系，而且一次编译需要大量的时间。

[1] https://www.gnu.org/software/make/

1.1 编译系统的概念

情景提要

- 当一个项目只有一个源文件，编译是容易的；
- 当一个项目有多个源文件，编译会变得复杂：编译有了依赖关系，而且一次编译需要大量的时间。

此时，可以使用编译系统来组织和优化编译。GNU Make[1] 是一种常见的编译系统。它使用特殊的语法来描述依赖关系，并且有着智能化的编译控制系统，只编译确实需要编译的文件。

[1] https://www.gnu.org/software/make/

1.2 编译系统的建模

可以把编译系统建模成有向图 $G = \langle V, E \rangle$。

- 每个需要编译的文件视作顶点。
- 边描述依赖关系: 若顶点 u 的编译需要用到顶点 v 的编译结果 $v.result$, 则 u 依赖于 v, 令图中存在 $u \to v$ 的有向边。

1.2 编译系统的建模

可以把编译系统建模成有向图 $G = \langle V, E \rangle$。

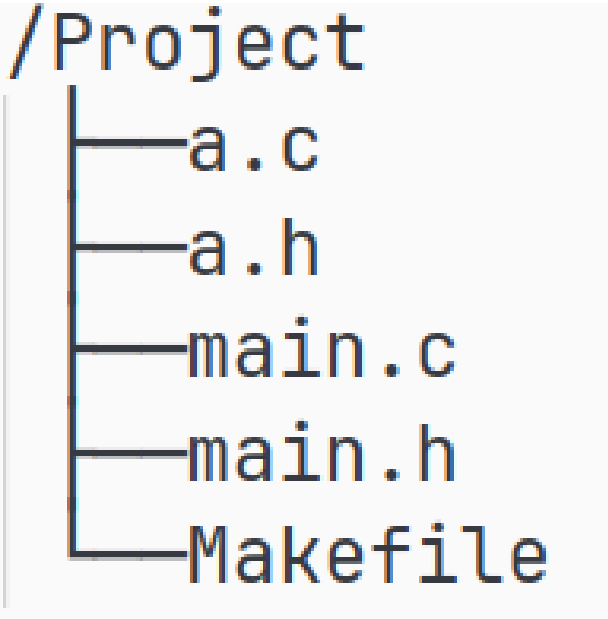
- 每个需要编译的文件视作顶点。
- 边描述依赖关系: 若顶点 u 的编译需要用到顶点 v 的编译结果 $v.result$, 则 u 依赖于 v, 令图中存在 $u \to v$ 的有向边。

若存在环 $v_1 \to v_2 \to \cdots \to v_m \to v_1$, 则对任意顶点（例如 v_1）, 它依赖于 v_2, 进而依赖于 $v_3, \cdots$, 依赖于 v_m, 最后依赖于 v_1 本身。

这是一个有向无环图 (Directed acyclic graph, DAG)。**在 DAG 上进行动态规划没有后效性**。

1.2 编译系统的建模

例如，对于以下项目：




```
1  # Makefile
2  main: main.o a.o
3      gcc main.o a.o -o main
4
5  main.o: main.c main.h
6      gcc -c main.c
7
8  a.o: a.c a.h main.h
9      gcc -c a.c
```

图：需要使用编译系统的项目

1.2 编译系统的建模

构造的图如下:

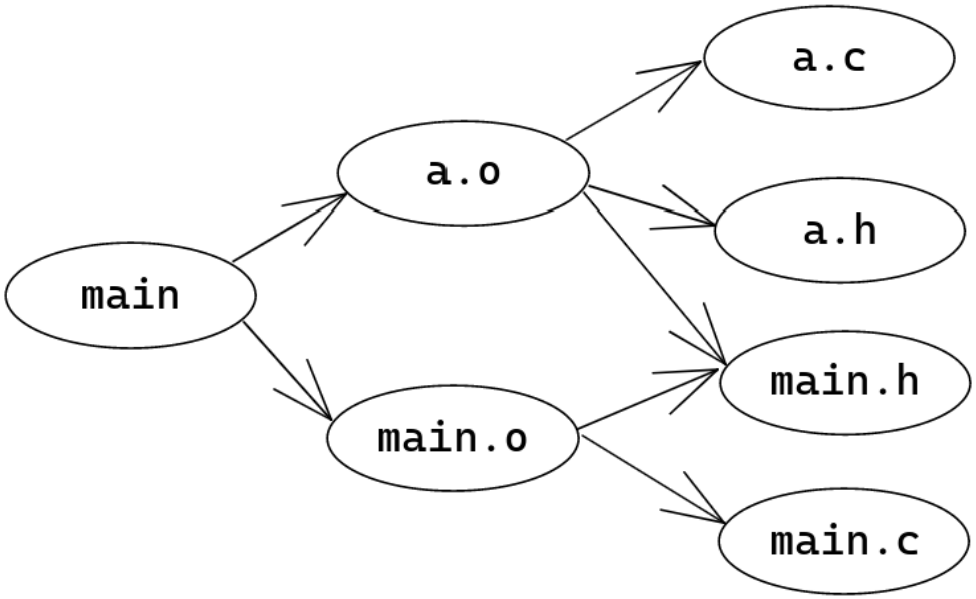


图: 编译系统的图论建模

1.3 GNU Make 的核心工作原理

作为编译系统，GNU Make 需要实现编译流程的自动化，最基本的就是能够自动分析依赖关系，自动编译。

1.3 GNU Make 的核心工作原理

作为编译系统，GNU Make 需要实现编译流程的自动化，最基本的就是能够自动分析依赖关系，自动编译。

自动编译的形式化描述

给出 DAG $G = \langle V, E \rangle$ 和 $u \in V$，给定函数 call_compiler_to_compile 以 u 和 $\{v.result : (u, v) \in E\}$ 为参数生成 $u.result$。需要生成 $u.result$。

1.3 GNU Make 的核心工作原理

1. 根据 Makefile 给出的依赖关系，构建出一个有向无环图。
2. 每次编译时，先依次编译它依赖的顶点（枚举出边），再编译它自己。
3. 使用动态规划可避免重复编译。此处记忆化搜索是高效的。

```
is_compiled = [0] * N

def make(u):
  if not is_compiled[u]:
    for v such that (u, v) in E:
      make(v)
    u.result = call_compiler_to_compile([v: for v such that (u, v) in E])
    is_compiled[u] = 1
```

图: GNU Make 的核心工作原理示意代码

1.4 GNU Make 的智能化编译控制

当某个顶点所有的依赖顶点的编译结果的时间戳都未比当前顶点编译结果的时间戳晚，那么说明当前顶点的编译结果已经是最新的了，不需要重新编译。

```
1  is_compiled = [0] * N
2
3  def make(u):
4    if not is_compiled[u]:
5      need_to_compile = 0
6      for v such that (u, v) in E:
7        make(v)
8        if u.result == None or v.result.time > u.result.time:
9          need_to_compile = 1
10     if need_to_compile:
11       u.result = call_compiler_to_compile([v: for v such that (u, v) in E])
12     is_compiled[u] = 1
```

图: GNU Make 的智能化编译控制示意代码

1.5 编译系统的本质

问题

为什么编译系统可以用有向无环图建模呢?

1.5 编译系统的本质

问题

为什么编译系统可以用有向无环图建模呢?

编译系统的本质是偏序关系!

偏序关系

S 上的偏序关系 $R \subset S^2$ 满足:

1. 反自反性: $x \not\prec x$;
2. 反对称性: $x \prec y \Rightarrow y \not\prec x$;
3. 传递性: $x \prec y \wedge y \prec z \Rightarrow x \prec z$。

1.5 编译系统的本质

问题

为什么编译系统可以用有向无环图建模呢?

编译系统的本质是偏序关系!

偏序关系

S 上的偏序关系 $R \subset S^2$ 满足:

1. 反自反性: $x \not\prec x$;
2. 反对称性: $x \prec y \Rightarrow y \not\prec x$;
3. 传递性: $x \prec y \wedge y \prec z \Rightarrow x \prec z$。

若 $x \prec y \Leftrightarrow$ 在图中存在 x 到 y 的路径, 即可用图表达偏序关系, 容易证明是无环的。**偏序关系的图论表示就是有向无环图!**

1.6 其他偏序关系

- 使用软件包管理器进行软件依赖管理。
- To-do list 进行代办事项的先后排序（e.g. 先做饭再吃饭）。
- 数学上，集合包含偏序，数论整除偏序，多维偏序等。
- 动态规划的状态转移。

1.6 其他偏序关系

- 使用软件包管理器进行软件依赖管理。
- To-do list 进行代办事项的先后排序（e.g. 先做饭再吃饭）。
- 数学上，集合包含偏序，数论整除偏序，多维偏序等。
- 动态规划的状态转移。

思考题

全序关系在图论中的表示是什么?

1.7 扩展

思考题 1

编译系统依赖管理和网络链接图、函数调用关系图有什么区别?

函数调用关系图: 描述函数之间的调用关系, 若 A() 调用了 B(), 建有向边 A $\rightarrow$ B。

1.7 扩展

思考题 1

编译系统依赖管理和网络链接图、函数调用关系图有什么区别?

函数调用关系图: 描述函数之间的调用关系, 若 A() 调用了 B(), 建有向边 A $\rightarrow$ B。

- 网页/函数可以递归地自己链接/调用自己, 或者递归地互相链接/调用, 形成的图不是有向无环图。
- 事实上, 形成的有向图没有任何限制。

1.7 扩展

思考题 2

说到没有任何限制的图，除了偏序关系，图还可以表达什么关系？

1.7 扩展

思考题 2

说到没有任何限制的图，除了偏序关系，图还可以表达什么关系？

可以表达一切二元关系！

- 有向关系用有向图（如依赖关系）；
- 双向关系用无向图（如社交网络中的好友关系）；
- 多种关系用带权图（如知识图谱中的不同种类的信息）。

主要内容

2.1 正则表达式和状态机介绍

正则表达式可以用来查找替换识别固定格式的字符串[2]

1. [nz]ju 可以匹配 nju, zju.
2. wt+qj 可以匹配 wtqj, wttqj, wtttqj, ...

[2] https://en.wikipedia.org/wiki/Regular_expression
[3] Problem Solving I, Nanjing University.

2.1 正则表达式和状态机介绍

正则表达式可以用来查找替换识别固定格式的字符串[2]

1. [nz]ju 可以匹配 nju, zju.
2. wt+qj 可以匹配 wtqj, wttqj, wtttqj, ...

可以使用有限状态自动机（DFA 或 NFA）来进行（基本的）正则表达式的匹配[3], 具体而说, NFA 的每一个状态 i 表示正则表达式的第 i 个字符。

[2] https://en.wikipedia.org/wiki/Regular_expression
[3] Problem Solving I, Nanjing University.

2.1 正则表达式和状态机介绍

有限状态自动机由五大部分组成:

1. Σ: 输入字母表;
2. S: 状态的非空有限集合;
3. $S_0 \subseteq S$: 初态集合;
4. $\delta : S \times \Sigma \to S$: 状态转移函数（NFA 允许 ε 转移）;
5. $F \subseteq S$: 终态集合。

2.1 正则表达式和状态机介绍

有限状态自动机由五大部分组成:

1. Σ: 输入字母表;
2. S: 状态的非空有限集合;
3. $S_0 \subseteq S$: 初态集合;
4. $\delta : S \times \Sigma \to S$: 状态转移函数（NFA 允许 ε 转移）;
5. $F \subseteq S$: 终态集合。

图论建模方法:

1. S 中的每一个状态, 对应于有向图中的一个顶点。
2. $\delta(u, \cdot)$ 描述了 u 的所有出边。

2.1 正则表达式和状态机介绍

DFA 匹配一个字符串 $s_0 s_1 \cdots s_m$，判断是否存在一个状态列表 $\langle u_0, u_1, \cdots u_{m+1} \rangle$，使得 $u_0 \in S_0, \delta(u_0, s_0) = u_1, \delta(u_1, s_1) = u_2, \cdots, \delta(u_m, s_m) = u_{m+1} \in F$。如果是 NFA，还允许 ε 移动。

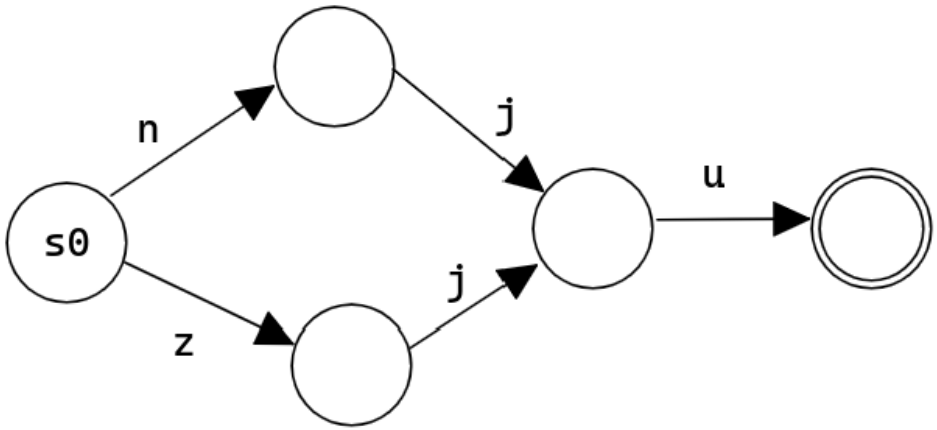
2.1 正则表达式和状态机介绍

DFA 匹配一个字符串 $s_0 s_1 \cdots s_m$，判断是否存在一个状态列表 $\langle u_0, u_1, \cdots u_{m+1} \rangle$，使得 $u_0 \in S_0, \delta(u_0, s_0) = u_1, \delta(u_1, s_1) = u_2, \cdots, \delta(u_m, s_m) = u_{m+1} \in F$。如果是 NFA，还允许 ε 移动。

人话：从 S_0 开始在图上按照字符串不停地走，看可不可以到 F。

2.1 正则表达式和状态机介绍

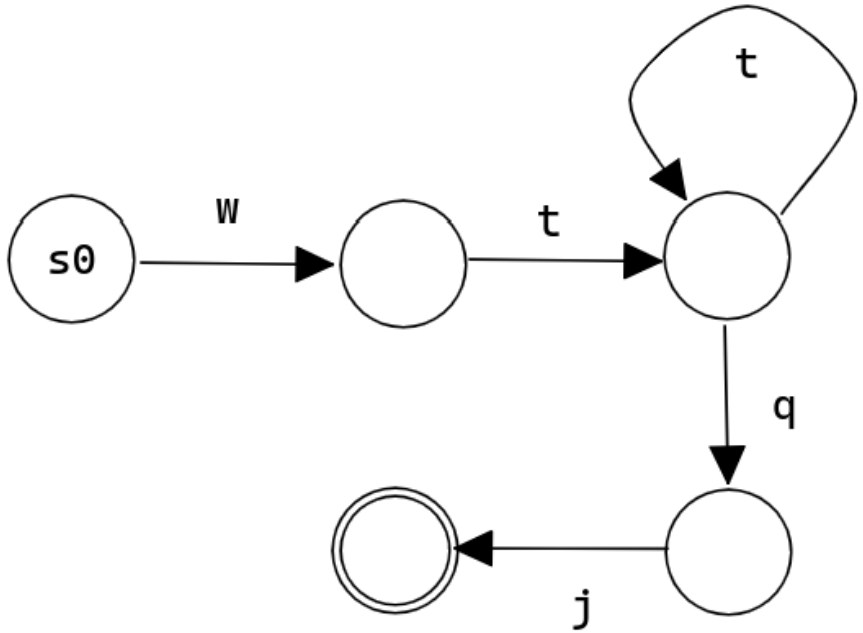
例如，[nz]ju 可以用以下状态机表示：



图：正则表达式 [nz]ju 的状态机

2.1 正则表达式和状态机介绍

例如，wt+qj 可以用以下状态机表示：



图：正则表达式 wt+qj 的状态机

2.2 自动机相关算法

- 在 NFA 上做匹配比在 DFA 上做匹配慢，有将 DFA 转化为 NFA 的算法（幂集构造[4]）。

[4] https://en.wikipedia.org/wiki/Powerset_construction
[5] https://en.wikipedia.org/wiki/DFA_minimization

2.2 自动机相关算法

- 在 NFA 上做匹配比在 DFA 上做匹配慢，有将 DFA 转化为 NFA 的算法（幂集构造[4]）。
- 为了节省空间优化性能，有最小化 DFA 的算法[5]。

[4] https://en.wikipedia.org/wiki/Powerset_construction
[5] https://en.wikipedia.org/wiki/DFA_minimization

2.2 自动机相关算法

- 在 NFA 上做匹配比在 DFA 上做匹配慢，有将 DFA 转化为 NFA 的算法（幂集构造[4]）。
- 为了节省空间优化性能，有最小化 DFA 的算法[5]。
- DFA、NFA 上可以进行动态规划。无后效性来于：考虑被匹配串的当前匹配位 j，则 j 时的状态只和 $j-1$（或之前）有关。

[4] https://en.wikipedia.org/wiki/Powerset_construction
[5] https://en.wikipedia.org/wiki/DFA_minimization

2.2 自动机相关算法

- 在 NFA 上做匹配比在 DFA 上做匹配慢，有将 DFA 转化为 NFA 的算法（幂集构造[4]）。
- 为了节省空间优化性能，有最小化 DFA 的算法[5]。
- DFA、NFA 上可以进行动态规划。无后效性来于：考虑被匹配串的当前匹配位 j，则 j 时的状态只和 $j-1$（或之前）有关。
  用 NFA 进行正则表达式的匹配即可使用动态规划，用 $f[i,j]$ 表示匹配字符串的前 i 个字符是否可以到达自动机的状态 j。

[4] https://en.wikipedia.org/wiki/Powerset_construction
[5] https://en.wikipedia.org/wiki/DFA_minimization

2.3 自动机相关的应用

- 现代计算机是 DFA，$\langle$存储器值1，存储器值2，...，存储器值$n\rangle$ 编码它的状态（存储器包括计数器、寄存器、Cache、RAM 等），而外界输入是匹配的字符串。

[6] https://en.wikipedia.org/wiki/Aho–Corasick_algorithm
[7] https://en.wikipedia.org/wiki/Suffix_automaton

2.3 自动机相关的应用

- 现代计算机是 DFA，$\langle$存储器值1，存储器值2，...，存储器值$n\rangle$ 编码它的状态（存储器包括计数器、寄存器、Cache、RAM 等），而外界输入是匹配的字符串。
- 在数字逻辑与计算机组成课里，时序逻辑电路中使用了 DFA 描述电路的切换。它也是一个 DFA。

[6] https://en.wikipedia.org/wiki/Aho–Corasick_algorithm
[7] https://en.wikipedia.org/wiki/Suffix_automaton

2.3 自动机相关的应用

- 现代计算机是 DFA，$\langle$存储器值1，存储器值2，...，存储器值$n\rangle$ 编码它的状态（存储器包括计数器、寄存器、Cache、RAM 等），而外界输入是匹配的字符串。
- 在数字逻辑与计算机组成课里，时序逻辑电路中使用了 DFA 描述电路的切换。它也是一个 DFA。
- 在字符串理论中，很多算法利用了一些特殊的自动机，如 Aho-Corasick algorithm[6]，Suffix automaton[7] 等。这些算法应用广泛。

[6] https://en.wikipedia.org/wiki/Aho–Corasick_algorithm
[7] https://en.wikipedia.org/wiki/Suffix_automaton

谢谢大家!