一种基于新型图模型的API推荐系统

吕 晨[1),2)] 姜 伟[3)] 虎嵩林[1)]

[1)](中国科学院计算技术研究所前瞻实验室 北京 100190)

[2)](中国科学院大学 北京 100149)

[3)](中国石油长城钻探工程有限公司测井技术研究院 北京 100101)

摘 要 对象实例化是软件工程类库复用中一个常见、复杂的问题.即根据已知源类型的对象,用户需要编写代码对目标类型进行实例化.研究人员提出了若干种API自动推荐系统辅助用户解决上述问题.然而这些系统不能同时兼顾准确率和查全率,因此在一定程度上影响了系统的实用性.该文提出了一种兼顾准确率和查全率,基于新型图模型的API推荐系统——APISynth.在查全率方面,APISynth使用一个新颖的全局图模型来表达类库中所有的API依赖关系和API历史使用信息.利用新的全局图模型具有的特殊Tag元素和可达性质以避免错误的API调用.在准确率方面,APISynth首先将对象实例化问题建模为Top-K子图查询问题,然后设计一种新的支持DAG形式解的图搜索算法,避免了传统的最短路径图搜索算法导致的查不准问题.实验结果表明,与现有多种方法相比,APISynth在准确率和查全率两方面均获得了较大提升.

关键词 代码辅助工具;API推荐工具;代码复用

中图法分类号 TP391 **DOI号** 10.11897/SP.J.1016.2015.02172

APISynth: A New Graph-Based API Recommender System

LV Chen[1),2)] JIANG Wei[3)] HU Song-Lin[1)]

[1)](*Advanced Research Laboratory, Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)

[2)](*University of Chinese Academy of Sciences, Beijing* 100149)

[3)](*R&D Academy of Well Logging, CNPC Greatwall Drilling Company, Beijing* 100101)

Abstract How to perform object instantiation is a common and complex problem in reusing an existing library. In this problem, given the object of source type, users need to write code to instantiate the destination type. Recently, researchers have proposed a variety of related API automatic recommender systems to fulfill this problem. These systems can assist users to solve the above issue in some degree. However, it is very difficult for such systems to achieve both good recall and precision, which affects the usability. This paper presents APISynth, a new graph-based API recommender system. In order to improve the recall, APISynth utilizes a new global graph model to give a better representation for the API dependencies in the relevant class library, and to include API historical usage information in. A special tag element is added to the new graph together with a new reach ability property to avoid false invocation of APIs. Meanwhile, unlike the shortest path problem proposed by the traditional methods, in order to improve the precision, APISynth models the object instantiation issue as a Top-K subgraphs searching problem. A new graph search algorithm is designed to directly support output of DAG-like solutions. Experimental results show that APISynth wins over the state of the art with respect to both the two criteria.

Keywords code assistant; API recommender; code reuse

收稿日期:2014-07-28;**最终修改稿收到日期**:2014-11-28.**本课题得到国家自然科学基金**(61070027)、**软件工程国家重点实验室**(SKLSE2012-09-02)**资助**.**吕 晨**,男,1983年生,博士研究生,中国计算机学会(CCF)会员,主要研究方向为软件工程、API可用性、服务计算.E-mail:lvchen@ict.ac.cn.**姜 伟**,男,1985年生,博士,主要研究方向为软件工程、API可用性、服务计算.**虎嵩林**(通信作者),男,1973年生,博士,研究员,主要研究方向为大数据处理、服务计算.E-mail:husonglin@iie.ac.cn.

1 引　言

随着计算机在人类生产、生活领域的不断普及，以及云计算、大数据等新兴计算模式的不断出现，以社交网站、搜索引擎为代表的超大型应用系统得到了迅猛发展. 在此背景下，计算机应用的复杂性不断增加，软件系统的规模也进入了爆发性增长的历史时期，软件开发和维护的代价一直在持续增大. 如微软的 Windows 操作系统的代码规模已经超过 4000 万行，多个版本的 Linux 操作系统的代码规模均在亿行以上，开源 Hadoop 的代码量达到了 230 万行①. 包括 Yahoo、Cloudra、Facebook 在内的数十家大型软件公司和近万名开发者构成了庞大的 Hadoop 社区，每年贡献的新增代码量都高达 30～40 万行②.

在这些超大规模、大型群体参与的软件系统开发过程中，类库复用可以有效地提高软件开发人员的编程效率、改善软件的质量[1]. 然而，由于类库中 API 的数量巨大并且使用方式复杂，类库复用的过程中往往也会涉及很多复杂的工作[2]. 对象实例化，作为其中的典型代表，就是类库复用中的一个突出的挑战性问题. 其根本目标是根据已知类型的对象获取未知类型的对象（或称为对未知类型的实例化）. 即给定起始点类型（Source Type）和终点类型（Destination Type）作为输入查询，输出从起始点类型到终点类型的方法调用序列[3].

面对规模和复杂性不断增长的软件系统，依靠手工查找、比对的方式实现对象实例化低效且易错. GEF（Graphical Editing Framework）作为一个常用 Eclipse 的图形交互类库，就涉及 35 万行代码③，共计 6 万多个 API 方法. 开发人员使用这个类库时，需要阅读海量的、甚至未及时更新的参考文档来找到一个正确的对象实例化方案. 据有关研究人员统计，编程人员查找和理解现有 API 用法的时间占用了超过 40% 的编程时间[4]. 因此，如果能够自动地完成对象实例化工作，为开发人员推荐最相关的方法调用序列，将极大地减少用户查找、理解、组合和调试的工作量，进而降低人工错误率，提高软件的质量. 因此，面向对象实例化的自动 API 推荐引起了诸多研究人员的关注[2-3,5-6]，已成为一个研究的热点[7]. 这些相关工作中实现的对象实例化工具[2-3,5-6]被称为 API 推荐系统.

这些 API 推荐系统[2-3,5-6]首先搜集查询（Source，Destination）涉及的类库源码或代码样例，然后采用本文称之为全局连通图[2]或局部非连通图[3,5-6]的模型对代码中包含的 API 依赖关系进行建模. 其中，前者对类库源码中所有 API 的依赖关系进行统一建模，而后者则分别对单个类文件或方法进行建模. 在图模型的支持下，二者采用基于图搜索[2-3,5]或模式匹配[6]的方法来生成从起始点类型（Source Type）到终点类型（Destination Type）的子图. 每个子图表示的方法调用序列就构成一个解.

Prospector[2]是一种基于全局连通图模型的方法. 它可以完整刻画类库中所有类型之间的转换关系，每一种可能的方法调用序列都可以从中找到. 理论上这种方式具有很好的查全率. 实际情况中，由于方法调用通常需要满足一个或多个输入参数的需求，方法调用序列常常会呈现为有向无环图（DAG）的形式. 然而，由于 Prospector 采用最短路径搜索算法，不能直接支持 DAG 图形式解的搜索，在搜索过程中会导致不少方法的输入参数未能被实例化. 为了解决这一问题，Prospector 将这些参数作为自由变量（free variable），要求开发人员手工输入新的查询搜索这些自由变量的实例化解，并组合这些解来生成最终结果. 可见，这种需要人工干预才能推荐出完整解的工作机制，会在一定程度上影响用户体验. 同时，Prospector 也无法对人工组合后的解进行精确的优劣排序，易导致用户选择无效解. 第 2 节将给出一个具体的示例对 Prospector 存在的问题予以说明. 本文的实验结果也表明该方法在准确性方面存在明显不足.

与之相比，文献[3,5-6]采用了基于局部非连通图模型的机制，它们的图模型包括多个局部非连通图，分别表达某个方法或类文件内部的 API 依赖关系. 单个局部图规模相对较小，能够大大地减少问题空间，因此有利于系统准确率的提升. 但是，这类方法的图模型不能表达孤立局部图之间的 API 依赖关系，难以对类库中所有的 API 依赖关系进行完整刻画，因此容易产生遗漏，造成系统查全率的下降. 尽管文献[6]提出一种查询分裂的技术可以实现跨多个局部图的搜索，但是它使用语句间的控制流构建图模型，无法对 API 之间的数据依赖关系进行刻画，从而

① http://www.ohloh.net/p/Hadoop
② http://hortonworks.com/blog/reality-check-contributions-to-apache-hadoop/
③ http://www.ohloh.net/p/eclipse-gef

使其能够描述的解空间变小，降低了系统的查全率.

表 1 对现有工作的特点进行了总结和分析. 由此可知，现有方法要么查全率不足，要么准确性无法保障. 为了解决这一问题，本文提出了一个新的全局连通图模型——加权 API 图，给出了新的不同于传统图论的可达性定义及搜索算法，并设计实现了一个新的 API 推荐系统——APISynth，来帮助编程人员完成对象实例化的任务.

表 1 对象实例化自动推荐方法的总结与对比

典型工具	技术特点						实验效果		问题分析
	图模型		搜索方式		API 依赖描述		准确率	查全率	
	全局	局部	路径搜索	跨图查询	数据依赖	控制依赖			
Prospector[2]	√	×	√	—	√	×	低	高	全局图模型上使用路径搜索
ParseWeb[5]	×	√	√	√	×	√	高	低	图模型无法描述数据依赖
XSnippet[3]	×	√	√	×	√	×	高	低	非全局图且不支持跨图查询
GraPacc[6]	×	√	×	×	√	√	高	低	非全局图且不支持跨图查询

本文的主要贡献包括：

(1) 提出一个新的加权全局连通图模型. 能够准确地表达相关类库包含的 API 依赖关系以及 API 历史使用信息. 与现有全局连通图模型不同，本文的图模型中节点表示 API 方法；边用于连接输入类型(接收者和方法的参数)与输出类型(方法的返回值)相匹配的方法，以此表示方法间的数据依赖，每条边带有一个参数标签(Tag)；API 历史使用信息(API 方法在相关项目中被使用的频率)用于对图模型进行完善和补充：将其作为服务质量 QoS (Quality of Service)赋值给相应的节点. 与经典图论不同，本文借助新元素 Tag 提出了新的可达性定义，以满足搜索 DAG 子图的需求.

(2) 提出一种新的图搜索算法. 借助新型图模型的结构及可达性质，提出了新的 Top-K 子图查询问题来建模方法调用序列的查询问题，并提出了关键路径松弛算法及 DAG 子图排序方法，指导编程人员选择正确结果.

(3) 通过一系列对比实验对所提出的方法进行系统化验证，实验结果表明，本文的方法获得了更高的查全率和准确率.

本文第 2 节介绍相关工作；第 3 节阐述加权 API 图模型；第 4 节对系统进行整体概述；第 5 节阐述算法细节；第 6 节讨论实验结果；第 7 节进行全文总结.

2 相关工作

本节首先综述对象实例化推荐工作的进展；然后介绍其他较为相关的工作.

2.1 对象实例化推荐

为便于讨论，本文根据图模型的异同，将现有方法分为两类：基于全局图模型的方法[2]和基于局部图模型的方法[3,5-6]. 需要指出，尽管 Nguyen 等人[6]将文献[6]归类为代码补全方法，但是由于该方法也可用于解决对象实例化的问题，本文也将其纳入本节一并讨论. 为了更直观地描述现有方法的特征，尤其是相关图模型的结构，以图 1 所示代码为例，本文在图 2 中给出各方法提出的图模型. 为便于表述，下文将使用代码的符号简记形式(图 1 右侧)进行相关问题的阐述.

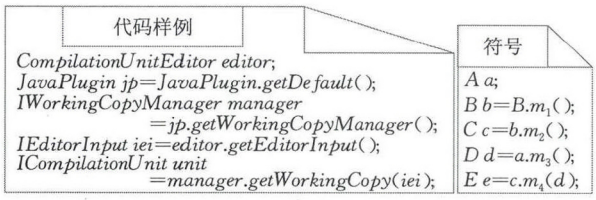


图 1 代码样例及其标记形式

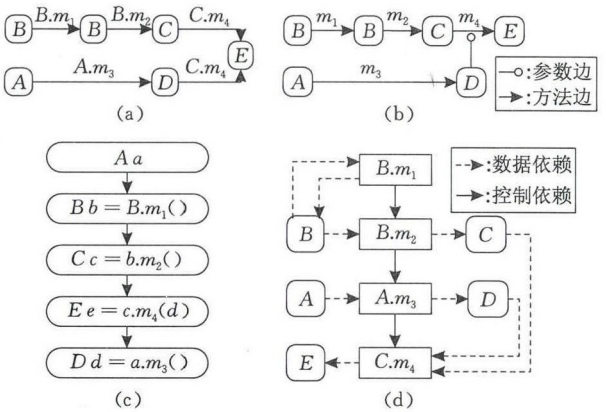


图 2 相关工作图模型示例(基于图 1 示例代码建模)

2.1.1 基于全局图模型的方法

Prospector[2]根据类库中的 API 依赖关系构建图模型. 图中的节点表示对象类型；边表示 API 方法调用，用来实现类型之间的转换. 图 2(a)展示了 Prospector 构建的图模型. 基于该图，Prospector 将对象实例化问题建模为一个传统的图搜索问题——

最短路径问题，并使用路径搜索算法求取起点类型到终点类型的方法调用序列. 这些方法调用序列在图中表现为路径或 DAG 的形式. 例如，当方法调用序列中存在带参数的方法调用时，该序列在图中将呈现为 DAG 的形式. 然而，Prospector 采用最短路径搜索算法不能直接求取 DAG 形式的解，因此需要额外的人工干预，具体过程见表 2.

表 2　Prospector 推荐过程示例

步骤	人工操作	结果
1	编程人员输入查询(*A*,*E*)，Prospector 在图 2(a)上搜索出路径〈*A*,*D*,*E*〉，返回结果：	*A a*； *D d*=*a*.*m*3()； ***C c***；//自由变量 *E e*=*c*.*m*4(*d*)；
2	编程人员手工识别出自由变量 *C*，并尝试输入查询(*A*,*C*)，(*D*,*C*)和(*Void*,*C*). Prospector 只能根据(*Void*,*C*)搜索出路径〈*B*,*B*,*C*,*E*〉，并返回结果：	*B b*=*B*.*m*1()； *C c*=*B*.*m*2()； *D d*； *E e*=*c*.*m*4(*d*)；
3	编程人员使用上述两次返回的结果，清除不必要的代码并组合生成最终解：	*A a*； *B b*=*B*.*m*1()； *C c*=*B*.*m*2()； *D d*=*a*.*m*3()； *E e*=*c*.*m*4(*d*)；
4	结果评分，考虑两个因素： (1) 初次搜索的路径的长度. (2) 该路径中自由变量对应分支路径长度的估计值(统一估计长度为 2.	

由表 2 可知，当满足查询的最终结果表现为 DAG 形式时，搜索过程中就会产生自由变量. 此时，Prospector 需要人工干预才能生成有效解. 但是过多的人工干预易造成错误操作，导致不正确结果的生成，从而降低了系统的准确率. 此外，Prospector 对结果的评价也不够精确和全面. 这一问题也存在于 DAG 图形式的服务组合问题中[8-10]. 本文称此问题为“弱 DAG 机制”问题. 已有的研究工作表明，带参数 API 的调用在实际应用中往往占有较大比例，例如在 3 个典型的 Java 项目(Eclipse 3.6.2，Tomcat 7.0，JBoss 5.0)中，带参数 API 方法的调用次数的平均占比为 59%[11]. 此外，本文对 GEF 类库的统计结果表明，带参数 API 方法的占比也达到 28%. 据此可知，对象实例化查询的解以 DAG 形式出现的概率较高，而由此引发的弱 DAG 机制问题会显著影响 Prospector 的准确率.

2.1.2　基于局部图模型的方法

基于局部图模型的方法[3,5-6]将相关的 API 依赖关系建模为多个图模型. 每一个图仅对应于一个特定单元的代码，例如类库中的一个类文件[3]或者代码样例中一个 API 方法包含的内部代码[5-6]. 这类方法构建的图模型由多个仅仅表达局部、特定单元 API 依赖关系的图组成，因此可称之为局部图模型.

文献[3,5-6]构建的局部图模型分别具有不同的特点. 例如，在文献[3]构建的图中，节点表示一条代码语句，边表示语句调用次序，如图 2(c)所示；文献[5]与文献[2]构建的图较为相似，不同点是前者在图中增加了专门指向方法参数的边，如图 2(b)所示；文献[6]构建的图中包括 3 类节点：数据节点(数据类型)、动作节点(方法调用)和控制节点(循环、分支)，同时使用数据依赖边和控制依赖边刻画节点间的关系，如图 2(d)所示.

上述工作采用的求解策略也具有差异性. 与文献[2]相似，文献[3,5]也将对象实例化问题建模为最短路径问题，并使用标准的图搜索算法在局部图模型上查找解. 与之不同，文献[6]则通过模式匹配的方式解决对象实例化问题：首先使用专门的工具 Grouminer[8]将局部图模型中频繁出现的子图抽取为模式；然后使用与查询匹配的模式来生成子图形式的解.

局部非连通图模型可以压缩问题空间，因此有利于提高系统的准确率. 但是这种非连通图无法表达图与图之间的关系，因此会遗漏掉部分有效解，从而降低系统的查全率. 有研究者可能会质疑，可通过构建跨局部图的 API 依赖连通局部图，从而提升系统的查全率. 然而，当这些局部图连通后，文献[3]所提出的模型将与全局连通图模型相似，仍会带来“弱 DAG 机制”的问题. 与文献[3]不同，文献[5-6]提出的每个孤立的局部图存在根据语句调用次序构建的边. 然而，在局部图模型中，图与图之间却不会存在语句调用关系. 因此文献[5-6]无法实现局部非连通图的“连通”. 此外，本文还注意到文献[5]提出了一种查询分裂的技术，能够实现跨图搜索. 但是这一技术的局限性在于跨图搜索的载体必须为根据语句控制流构建的图模型. 但是由于这类图模型无法表示 API 之间的数据依赖关系，其能够描述的解空间有限. 因此，即使采用查询分裂技术，文献[5]的查全率依然有提升空间. 本文在实验部分对此进行了验证.

综上，现有工作难以兼顾准确率和查全率.

2.2　其他相关工作

文献[12-20]根据用户的需求进行代码推荐，是与本文较为相关的研究工作.

CodeBroker[12]和 Strathcona[13]首先分析用户

正在编辑的代码来抽取上下文信息;然后从代码库中查找与上下文相似的代码样例.然而,这些方法的输出仅仅是与上下文信息相似的代码样例,而不是满足对象实例化查询的方法调用序列.

文献[14-18]利用数据挖掘技术推荐代码.这些方法采用关联规则挖掘[14-16]、随机游走[17]、时序模式挖掘[18]等策略推荐代码.但是,这些方法的输入不支持对象实例化查询,因此不能生成满足要求的方法调用序列.

文献[19-20]采用信息检索技术搜索样例代码.通常,这些工作首先对代码库中的源文件进行索引;然后通过信息检索技术(如关键词匹配)来查找相关的代码样例.文献[19]对 13 个典型的代码搜索引擎(如 Codease①, Google Code Search②, Krugle③)进行了对比.然而,代码搜索引擎常返回海量、结构复杂的结果页面,使用户很难快速地定位正确解.

上文介绍了相关工作,接下来阐述本文为刻画类库 API 依赖关系所构建的图模型,本文设计的搜索算法将根据此图进行求解.

3 加权 API 图模型

3.1 相关术语介绍

本文使用的相关术语及其定义见表 3.

表 3 相关术语

术语	定义
方法(m_i)	m_i是一个三元组$\{I,o,QoS\}$.I 是 m_i的输入参数类型集合,包括 m_i的接收者类型和参数类型;o④ 是 m_i的输出参数类型,即 m_i的返回值类型;QoS 是指 m_i的服务质量.
服务质量 QoS	QoS 是 m_i的非功能性属性.即 API 方法的使用频率.
查询 R	形式为($Source$,$Destination$),其中 $Source$ 为起始类型,记为 $R.S$,是用户已知对象的类型;$Destination$ 为终类型,记为 $R.D$,是用户需要实例化的类型.
方法匹配	给定两个方法 m_a和 m_b,当 m_a的输出类型与 m_b的一个输入类型匹配时,称 m_a匹配 m_b.
全局服务质量 $GQoS$	方法调用序列中全部方法的服务质量按照特定规则计算的聚合值(见 3.5 节).

3.2 WAG 图模型

为准确地表达相关类库包含的 API 依赖关系以及 API 历史使用信息,本文设计了一个新型的加权 API 图模型(Weighted API Graph, WAG)$G=(M,E)$对 API 依赖关系进行建模,如图 3 所示.本文将基于此图来搜索满足对象实例化查询的解.在 WAG 中,节点集 M 表示 API 方法集合,$\forall m_k \in M$, $m_k=(k,I,o,\omega)$.其中,k 是节点 m_k的标识,I 和 o 分别表示 m_k对应方法的输入参数类型集合和唯一的输出参数类型.m_k有相应的权值(即方法的 QoS),由加权函数 $\omega: M \to R$ 给出.有向边集 E 表示方法匹配集合,该集合满足:$\forall e_k \in E, e_k=(m_u, m_v, tag_{e_k})$.其中,$m_u$和 m_v分别是边的头结点(head node)和尾节点(tail node),m_u是的 m_v直接前继,m_v是 m_u的直接后继,m_u对应的方法匹配 m_v对应的方法.e_k的标记 tag_{e_k}满足:

(1) $tag_{e_k}=m_u.o$;

(2) $tag_{e_k} \in m_v.I$.

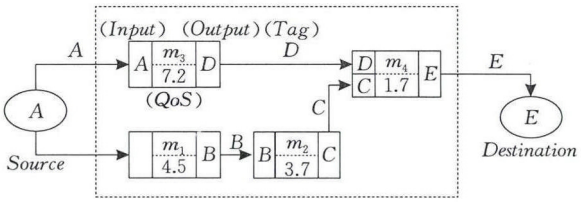


图 3 加权 API 图(基于图 1 示例代码建模)

需要特别指出,当查询请求 R 发起时,图中将动态生成临时的起始节点 $Source$ 和终节点 $Destination$,它们满足:

(1) $Source.I=\varnothing \wedge Source.o=R.S$;

(2) $Destination.I=R.D \wedge Destination.o=$ null.

此外,WAG 也可以处理下述两类特殊情形:情形 1(向上转型):在面向对象的编程语言中(如 Java),因为向上转型(即将子类对象转换为父类对象)为永远安全的操作,所以尽管有些向上转型操作未在相关源代码中出现,本文也将在图中构建表示这类操作的节点;情形 2(静态方法):由于调用静态方法无需输入接收者对象,当查询请求发起时,将构建从起始点到静态方法对应节点的边.

3.3 图性质

与经典图论的节点可达性不同,WAG 图的节点可达性如下.

定义 1. 节点的可达性.给定 WAG 图 $G=(M,E)$,在一次从起始节点 $Source$ 开始的搜索过程中,$\forall m_i \in M$, m_i可达当且仅当 $\forall input_i \in m_i.I$, $\exists p_i=\langle Source, \cdots, m_{i-1}, m_i \rangle$,使得路径 p_i中的边 $e=(m_{i-1}, m_i, tag_e)$满足 $tag_e=input_i$.

① http://www.codase.com
② https://code.google.com
③ http://www.krugle.com
④ 在面向对象的编程语言(如 Java)中,API 方法可能包含多个输入参数,但只有一个返回值.因此,本文使用大写字母 I 表示包含输入参数的集合,而使用小写字母 o 表示单个返回值类型.

在定义 1 中，一个节点 m_i 可达，不仅要求从 $Source$ 到 m_i 存在路径，而且要求存在从 $Source$ 到 m_i 的路径集合，使得该路径集合必须包含 m_i 的全部输入类型作为 Tag 的入边. 这不同于经典图论的节点可达性. 在经典图论中，判断一个节点 m_i 是否可达，只需判断从 $Source$ 到 m_i 是否存在路径即可.

基于 WAG 图的搜索算法在访问节点时遵循如下原则：若一个节点 m_i 可达，则可访问 m_i 的后继节点；否则，不能访问. 特别地，当在 WAG 图上进行搜索时，可访问 m_i 的后继节点的含义为：可调用 m_i 对应的 API 方法获取 m_i 输出类型的对象. 因此，当 m_i 输入类型的对象还未被全部获取时，定义 1 中的可达性能够避免错误地调用该方法获取其输出类型的对象. 例如，在图 3 中，当从 $Source$ 开始搜索时，首先找到路径 $p=\langle Source, m_1, m_2, m_4\rangle$. 然而，此时 m_4 并不可达. 原因在于 m_4 的另外一个输入类型 D 作为 Tag 的入边未被 p 包含.

需要指出，在传统的全局连通图模型中，其节点的可达性质与经典图论相同：对于任意节点 m_i，当存在从起始节点到 m_i 的路径时，m_i 即为可达. 它不能保证被调用的 API 方法的所有输入类型均已被实例化. 例如，在图 2(a)中，当从节点 C 开始搜索时，尽管可通过访问边 $C.m_4$ 使得节点 E 可达，但是此时 API 方法 $C.m_4$ 的另外一个输入 D 还未被实例化，从而将导致自由变量的出现. 根据第 1 节中的分析可知，这将引发“弱 DAG 机制”问题，因此会显著影响相关系统的准确率.

综上所述，与现有全局连通图模型相比，本文提出的 WAG 模型能够更准确地描述 API 方法能否被调用，因此可以基于 WAG 及其节点可达性质设计新的搜索算法来提高系统的准确性. 可见，在对象实例化领域，加权 API 图模型的提出弥补了现有全局连通图模型的不足，是本文的一个主要创新.

3.4　问题定义

给定查询请求 R，本文将其对应的对象实例化问题转化为基于 WAG 模型的 Top-K 子图查询问题.

定义 2.　Top-K 子图查询问题. 给定查询请求 R 和 WAG 图 $G=(M,E)$，加权函数 $\omega: M\rightarrow R$ 表示从节点集合到实数集合的映射. 子图集合 SG_{All} 表示所有满足 R 的方法调用序列对应子图的集合. SG_{All} 中的每个子图定义了其蕴含的方法 $(m_1, m_2, \cdots, m_n)$ 的一个调用结构，这些方法满足如下关系：

(1) $m_i.I \subseteq \bigcup_{j=1}^{i-1} m_j.o \cup R.S$；

(2) $R.D \subseteq \bigcup_{j=1}^{n} m_j.o$；

子图 SG_i 的权重值是指 SG_i 包含节点的权重值的聚合值：$\omega(SG_i)=SG_i.GQoS$. 定义最优子图 SG 的 $GQoS$ 为 $\delta=Min\{\omega(SG_i)\mid SG_i\in SG_{ALL}\}$. SG_{All} 中的最优子图定义为权重 $\omega(SG_i)=\delta$ 的任何子图.

Top-K 子图集合 $SG_{\text{Top-}K}$ 表示 SG_{All} 中权重值最优的 K 个子图组成的集合，它们满足：

(3) $Max\{\omega(SG)\mid SG\in SG_{\text{Top-}K}\}\leq Min\{\omega(SG')\mid SG'\in SG_{\text{All}}-SG_{\text{Top-}K}\}$.

由上述定义可知，在 WAG 图中找出满足查询请求 R 的前 K 个 GoS 最优子图的问题等价于 Top-K 子图查询问题. 本文将在第 5 节详细阐述 Top-K 子图查询问题的图搜索算法. 此外，本文的目标子图是一个个 DAG 形式的解，只有在特殊情况下才是链状解(子图中所有方法的输入参数均为一个).

例 1.　以图 3 为例，给定对象实例化查询 (A,D)，图中矩形框内的图可作为一个 DAG 形式的解.

3.5　QoS 类型与计算规则

WAG 中子图 $GQoS$ 的计算规则将在本小节进行说明. 该计算规则与 QoS 类型和调用模式相关.

3.5.1　QoS 类型

文献[21-22]定义了两类 QoS：(1) 否定型，即 QoS 值越大，服务质量越差，比如响应时间和价格；(2) 肯定型，即 QoS 值越大、服务质量越好，比如吞吐量和声誉. 同时，为了对多个 QoS 进行统一度量，按照度量方式的不同又可把 QoS 分为以下 4 种类型：(1) 累加型，如使用次数，两个 API 总的使用次数由两个 API 各自的使用次数累加得到；(2) 最小值型，如吞吐量，两个顺序调用的 API 处理数据的全局吞吐量，由具有最小吞吐量的 API 决定；(3) 乘积型，如声誉、可靠性；(4) 最大值型.

在本文中，方法的 QoS 将利用 API 方法在相关项目中的使用次数进行计算. 式(1)给出了具体的计算方式：

$$m_i.QoS=\begin{cases} Log\dfrac{Max-Min}{M_i.Count-Min}, & m_i.count>Min \quad (1)\\ +\infty, & m_i.count=Min \quad (1') \end{cases}$$

在式(1)中，$m_i.Count$ 表示方法 m_i 的在相关项目中被使用的次数，Max 和 Min 分别表示所有方法中，被使用的最大次数和最小次数. 本质上，式(1)计算出的 QoS 值是方法调用次数的正则化形式. 因为

方法的 QoS 越小表明方法被使用的频率越高，被采纳的可能性也越高，所以 QoS 越小的方法服务质量越优，反之越差. 因此，本文设计的 QoS 属于否定型.

3.5.2 $GQoS$ 计算规则

$GQoS$ 的计算规则由图的组合模式和 QoS 类型共同决定. 在组合模式方面，WAG 模型中包含的原子结构可被顺序、合并和分叉 3 种组合模式充分描述，如图 4 所示. 在 QoS 类型方面，本文定义的 QoS 表示 API 的调用次数，适用于累加型度量标准. 据此可定义相关组合模式的 $GQoS$ 计算规则，如表 4 所示.

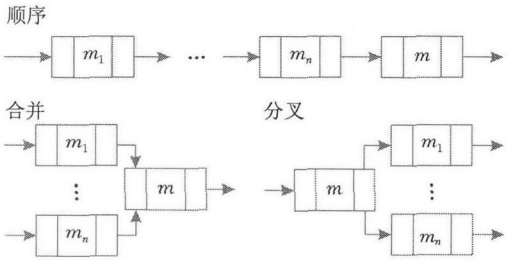


图 4 DAG 的组合模式

表 4 *GQoS* 的计算规则

模式	计算规则
顺序	$m.GQoS=m.QoS+\sum_{i=1}^{n} m_i.QoS$
合并	$m.GQoS=m.QoS+\max\{m_i.GQoS\}\|_{i=1}^{n}$
分叉	$m_k.GQoS=m.GQoS+m_k.QoS(1\leq k\leq n)$

在顺序模式中，因为节点依次被顺序调用，所以采用累加型函数计算 $GQoS$ 聚合值. 即最后一个被调用节点 m 的由其所有前驱的 QoS 和其自身的 QoS 累加得到.

在合并模式中，当 $m_1 \neg m_n$ 均已能够被调用时，才可调用节点 m. 在本文中，m 的 $GQoS$ 由 $m_1 \neg m_n$ 中 $GQoS$ 质量最差的节点决定，同时又因为节点的 QoS 属于否定型，所以采用最大值函数和累加型函数计算 $GQoS$ 聚合值. 即合并节点 m 的 $GQoS$ 是所有前驱 $GQoS$ 的最大值与其自身 QoS 之和.

在分叉模式中，当 m 能被调用时，则可调用其分支节点 $m_1 \neg m_n$ 中的任一节点. 因此可采用累加型函数计算 $GQoS$ 聚合值，即分支节点的 $GQoS$ 由其分叉节点 m 的 $GQoS$ 和其自身的 QoS 累加得到.

例 2. 以图 3 为例，利用表 4 中的 $GQoS$ 计算规则可得

$$\begin{aligned} m_4.GQoS &= \sum(\max\{m_2.GQoS, m_3.GQoS\}, m_4.QoS) \\ &= \sum(\max\{(m_1.QoS+m_2.QoS), m_3.QoS\}, \\ &\quad m_4.QoS) \\ &= \sum(\max\{(4.5+3.7), 7.2\}, 1.7) \\ &= \sum(\max\{8.2, 7.2\}, 1.7) \\ &= 8.2+1.7=9.9. \end{aligned}$$

需要指出，WAG 图中任意子图 SG_i 的 $GQoS$ 是根据 SG_i 中所有节点 QoS 按照指定规则计算出的聚合值. 该值最终等于 SG_i 中终结点的 $GQoS$.

4 APISynth 系统概述

本文设计并实现了 APISynth 系统，该系统实现了 Top-K DAG 图搜索算法. 如图 5 所示，APISynth 包括图构建模块、解搜索模块和结果排序模块.

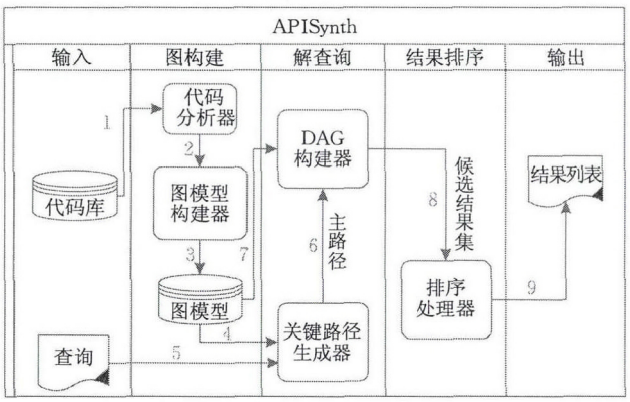


图 5 APISynth 设计架构图

图构建模块负责分析类库源码并构建 WAG 模型，包括代码分析器和图模型构建器. 代码分析器负责分析源代码，生成 API 依赖信息和 QoS 信息. 软件分析技术可分为静态分析和动态分析两类[22].

借助工具 Spoon① 和 Javassit②，本文对类库源码进行了静态分析，抽取其中的 API 依赖关系. 图模型构建器则负责构建 WAG 模型.

解搜索模块完成 Top-K 子图查询，包括关键路径搜索器和 DAG 构建器. 前者负责生成关键路径. 后者负责构建每一条关键路径对应的 DAG.

结果排序模块使用排序处理器对推荐结果进行排序.

下文将分别对 3 个模块中的工作进行介绍.

4.1 WAG 图的存储结构及构建

鉴于 WAG 图的特殊结构和新的节点可达性质，为提高搜索算法的效率，本文未使用经典的图存储结构，如邻接表或邻接矩阵表示 WAG，而是设计了新型的存储结构，如图 6 所示，包括节点表，倒排索引表和可触发参数表.

① http://spoon.gforge.inria.fr/
② http://sourceforge.net/projects/jboss/files/Javassist

节点表

Node	Input	Output	*selfQoS*	*GQoS*	*Number*	*Status*
m_1	*A*	*B*	4.5	4.5	1	Disabled
m_2	*B*	*C*	3.7	8.2	1	Disabled
m_3	*A*	*D*	7.2	7.2	2	Disabled

倒排索引表

Input	*NodesList*
A	m_1, m_3
B	m_2
C	m_4
D	m_4

可触发参数表

Input	*GQoS*	*Provider*
A	0	*Source*
B	4.5	m_1
C	8.2	m_2
D	7.2	m_3

图 6　图 3 所示 WAG 对应的部分数据结构

节点表用来存储节点及其相关信息. 本文使用五元组表示节点，具体形式为 $\{m_i, I_{mi}, O_{mi}, selfQoS, GQoS, Number, Status\}$. 其中，$m_i$ 是节点的标识；I_{mi} 和 O_{mi} 分别表示节点的输入参数集和输出参数集；$selfQoS$ 表示节点自身的 QoS；$GQoS$ 表示从起始节点到当前节点的全局 QoS；$Number$ 用于记录节点需要被实例化的输入参数的个数，初始值为节点输入参数集的大小；$Status$ 表示节点的可达状态，默认状态为不可达，记为 *disabled*. 在搜索过程中，当 $Number$ 变为 0 时，将 $Status$ 置为 *enabled*，表示节点可达. 需要指出，$GQoS$、$Number$ 和 $Status$ 将在搜索过程中动态更新(见第 5 节).

倒排索引表用来表示和存储 WAG. 表中的项为键值对(参数，节点集). 其中，键为参数，其值为需要该参数作为输入的节点集合. 倒排索引表的使用能极大提高 Top-K 子图的搜索效率. 尽管与传统的邻接表和邻接矩阵相比，使用倒排索引表存储图将耗费更多的内存空间，然而这对主流计算机并不构成挑战性问题. 本文将在 6.3.3 小节对此进行讨论.

可触发参数表(Reachable Precondition Table，RPT)用来存储搜索过程中可达节点的输入参数及其最优提供者的相关信息.

4.2　Top-K 子图搜索

本小节阐述 Top-K DAG 图搜索算法，它能直接求取 DAG 形式解，而不仅仅是传统最短路径算法求取的链状解. 因此，基于 Top-K DAG 搜索算法的 APISynth 系统无需人工干预，可自动地完成对象实例化解的推荐.

对于 DAG 形式的子图，本文作者观察到其包含一条特殊路径 p，满足 $p.GQoS = DAG.GQoS.p$ 被称为次 DAG 的关键路径. 受此启发，算法首先生成关键路径；然后基于关键路径构建 DAG 图形式的解. 因此，Top-K 子图查询问题简化为求取 Top-K 条关键路径. 根据以上分析，本文需要解决如下问题：

(1) 如何在未知 DAG 的情况下求取关键路径？

(2) 如何根据关键路径构建 DAG，并保证 DAG 全局服务质量的最优性以及 DAG 中的节点均满足定义 1 中的可达性质？

(3) 如何求取更多条关键路径以获得 K 个 $GQoS$ 最优的 DAG？

Top-K DAG 搜索算法通过 4 个主要步骤解决上述问题，如图 7 所示(算法细节见第 5 节).

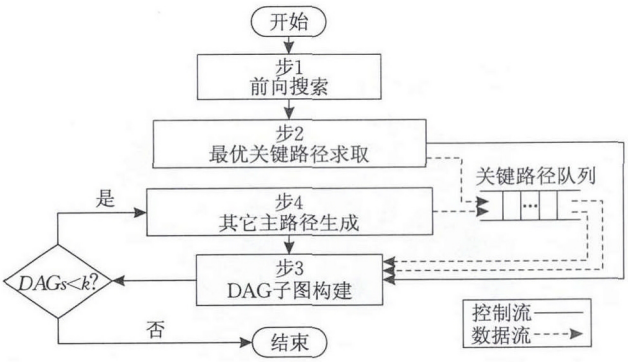


图 7　Top-K DAG 形式的解的搜索流程

(1) 执行一次从 $Source$ 节点开始的前向搜索，在搜索过程中确认所有可达的节点，并记录可达节点相关的信息，如 $GQoS$ 和节点输入的最优提供者.

(2) 根据步 1 中记录的信息，执行一次从 $Destination$ 节点开始的后向搜索，求取出 $GQoS$ 最优的关键路径 kp_{opt}，并将其放入优先队列中.

(3) 弹出优先队列中的关键路径，并利用其构建 DAG. 如果 DAG 的数量小于设定的阈值，则执行步 4；否则，终止.

(4) 对当前弹出的关键路径使用“松弛”操作来生成其他 $GQoS$ 较差的关键路径. 然后返回步 3 以生成更多的 DAG.

4.3　结果排序

为减少开发人员检查结果的次数，需要对候选结果进行排序，将优越的结果推荐给开发人员. 本文采用 4 个度量标准对结果进行评价，包括 $GQoS$、节点个数、边数和直径(关键路径的长度)①.

本文使用的排序准则为上述 4 个度量值越小，结果排名越高. 其合理性体现为：(1) $GQoS$ 反映了一个子图对应方法调用序列的总体使用频率，而优先选择较常使用的代码是编程人员的共识[3]；(2) 其余

① 可根据需要将 4 个度量标准扩展至更多个.

3 个度量标准反映了一个子图对应代码的数量(度量值越小,代码量越小).据统计,编程人员倾向使用代码量小的方案[2-3].

传统方法[2-3,5-6]仅采用路径长度来评价结果的代码长度.与之不同,本文综合考虑了待评价结果中包含的节点、边和直径等信息,因此可令 DAG 图形式解的优劣得到更全面、准确地反映.

本质上,此处的排序问题是一个多对象多属性的排序问题.这与数据库领域的多维排序类似.因此,本文的排序算法可借鉴经典的多维数据集排序算法——Fagin 算法[23].算法流程如图 8 所示.

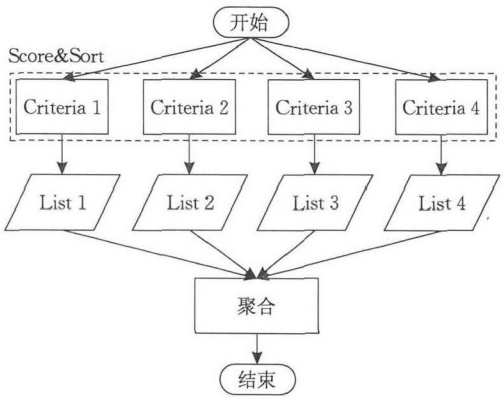


图 8 APISynth 的排序算法流程

首先,计算DAG形式解的4个度量值(Criteria 1～Criteria 4),并分别根据不同的度量标准对结果进行排序(为提高效率,该过程可并行执行).式(2)定义了 DAG 图形式解 SG_i 度量值的计算规则:

$$SG_i.Score=\begin{cases}\dfrac{SG_i.C_k-Min\{C_k\}}{Max\{C_k\}-Min\{C_k\}}, & Max\{C_k\}>Min\{C_k\} \quad (2)\\ 1, & Max\{C_k\}=Min\{C_k\} \quad (2')\end{cases}$$

其中,$C_k(k\in 1,2,3,4)$表示相应的度量标准的得分.$Max(C_k)$和 $Min(C_k)$分别表示所有结果中 C_k 度量标准的最大得分和最小得分.

其次,生成各度量标准的评分表(List 1～List 4),并计算聚合值得到最终评分:

$$SG_i.agg=\sum_{k=1}^{4}\omega_k\cdot SG_i.C_k \quad (3)$$

其中,权值 ω 满足$\sum_{k=1}^{4}\omega_k=1$.

5 Top-K DAG 搜索算法详解

本节详细阐述 Top-K DAG 搜索算法.它包括 3 个核心模块:(1)最优关键路径求取①;(2)如何根据关键路径构建 DAG;(3)如何通过对当前关键路径进行松弛操作,生成新的关键路径.

5.1 最优关键路径求取(步 1～2)

算法的前两步旨在求取 $GQoS$ 最优的关键路径.为此,本文实现了 Sim-Dijkstra 算法.算法的基本思想是:(1)根据查询请求,执行一次从端点 *Source* 到端点 *Destination* 的前向搜索,找出两端点之间所有可达的节点;(2)将每个可达节点输入的最优提供者记录到可触发参数表 RPT 中;(3)当前向搜索结束后,发起一次后向搜索过程 Backward-KP 求取最优关键路径.

5.1.1 相关概念

定义 3. 节点输入的最优提供者.即该节点所有可达前驱中,提供该输入且 $GQoS$ 最小的前驱结点.其形式化定义如下:给定 WAG 图 $G=(M,E)$,$\forall m_i\in M$,$\forall input_i\in m_i.I$,$input_i$ 的最优提供者为 p_{opt},$p_{opt}\in M$,当且仅当 p_{opt} 满足:

(1) $p_{opt}.status=enabled$;

(2) $p_{opt}.o=input_i$;

(3) $p_{opt}.GQoS=Min\{m.GQoS\mid m\in M\wedge m.o=input_i\wedge m.status=enabled\}$.

定义 4. 关键前驱.即某节点所有输入的最优提供者中 $GQoS$ 最大的那个最优提供者.其形式化定义如下:给定 WAG 图 $G=(M,E)$,$\forall m_i\in M$,节点 M_i 的关键前驱定义为 $\pi(m_i)=m_{kp}$,当且仅当节点 m_{kp} 满足:

(1) $m_{kp}\in M$;

(2) $m_{kp}.o\in m_i.I$;

(3) $m_{kp}.GQoS=Max\{input_i.p_{opt}.GQoS\mid input_i\in m_i.I\}$.

例 3. 以图 3 为例,对于 m_4 的两个输入 C 和 D,它们对应的最优提供者分别为 m_4 的前驱 m_2 和 m_3.并且有

$(m_2.GQoS=4.5+3.7=8.2)>(m_3.GQoS=7.2)$.

由此可知,m_2 是 m_4 的关键前驱.

定义 5. 关键路径.即 DAG 中一串顺序相连的关键前驱构成的路径.其形式化定义如下:在 WAG 图 $G=(M,E)$ 中,给定从 m_i 节点到 m_j 节点的子图 SG,则 SG 的关键路径为 $p=\langle m_i,m_{i+1},\cdots,m_j\rangle$,当且仅当 p 中节点满足:

$$\pi(m_{k+1})=m_k,\ i\le k\le j-1.$$

例 4. 图 9 展示了一个 DAG 及其关键路径.

① 与最优 DAG 对应的关键路径为最优关键路径.

需要指出，关键路径不是一条随意选取的普通路径，它的 $GQoS$ 与 DAG 的 $GQoS$ 相同.

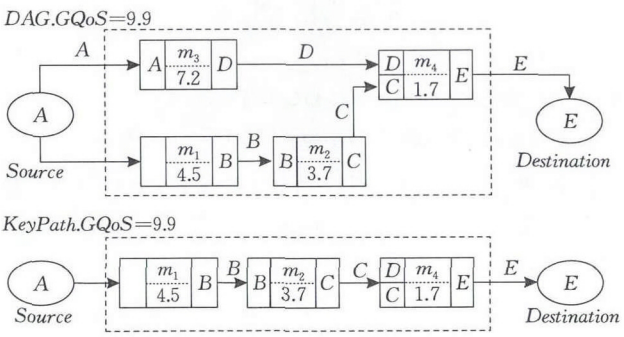


图 9　DAG 及其关键路径示例

5.1.2　Sim-Dijkstra 算法

Sim-Dijkstra 算法使用一个优先队列 *enabledNodes* 存储未处理的可达节点. 队列 *enabledNodes* 的性质是优先弹出具有最小 $GQoS$ 的节点. 其巧妙之处在于能够避免可达节点的多次更新处理，提高算法效率. 具体过程见算法 1：首先，使用 *Source* 节点来初始化 *enabledNodes*（第 1～2 行）；其次，使用 $popBest()$[①]操作优先弹出 *enabledNodes* 中 $GQoS$ 最好（最小）的节点. 然后，对于每个 *enabledNodes* 中弹出的节点 v，如果 v 的输出（par）未被存储到 RPT 中，算法将添加相关的项（$par, v.GQoS, v$）到 RPT 中（第 4～7 行）；最后，对于 v 的每个后继节点 u，令其 *Number* 值减 1，以表示这些节点的相关输入已被实例化. 因此，当 u 的 *Number* 值为 0 时，其状态将被置为可达（第 8～11 行）. 上述操作会不断循环执行以获得新的可达节点并将它们存储到 *enabledNodes* 中. 当 *enabledNodes* 为空时，算法将终止.

需要指出，Sim-Dijkstra 算法能够确保每一个 RPT 表项中参数（par）的提供者（$provider$）均为其最优提供者. 究其原因在于优先队列 *enabledNodes* 中 $GQoS$ 最好的节点 p_{opt} 总是被优先处理. 因此，p_{opt} 输出参数 M 将优先存入 RPT 表中. 当其他提供参数 M 的节点再被处理时，根据算法 1 第 5 行中的判断逻辑，这些节点将不能作为 M 的提供者存入 RPT 表项中. 综上所述，当算法 1 结束后，RPT 表中所有参数的提供者均为其最优提供者.

算法 1.　Sim-Dijkstra 算法.

输入：WAG 图模型、优先队列：*enabledNodes*

输出：节点可达信息（RPT 表项）

```
1.  初始化优先队列 enabledNodes;
2.  enabledNodes.add(Source);
3.  WHILE enableNodes≠∅ DO
4.    v⇐enableNodes.popBest();
5.      IF v.o∉RPT.keys() THEN
6.        FOREACH node u∈v.successors DO
7.          RPT.addentry(par,v.GQoS,v);
8.          u.Number--;
9.          IF u.Number=0 THEN
10.           u.GQoS⇐getGQoS(u);
11.           enabledNodes.add(u);
12.         ENDIF
13.       ENDFOR
14.     ENDIF
15. ENDWHILE
```

例 5.　以图 3 为例说明 Sim-Dijkstra 算法的执行过程，见表 5.

表 5　Sim-Dijkstra 算法执行步骤

步骤	*enabledNodes*（$GQoS$）	RPT 项参数（$GQoS$，$provider$）
1	*Source*(0)	
2	m_1(4.5)，m_3(7.2)	A(0，*Source*)
3	m_2(8.2)	B(4.5，m_1)；D(7.2，m_3)
4	m_4(8.2)	C(8.2，m_2)
5	*Destination*(9.9)	E(9.9，m_4)

5.1.3　Backward-KP 过程

利用 Sim-Dijkstra 算法输出的 RPT 表，本文采用 Backward-KP 过程生成所需关键路径，具体过程见算法 2：首先，依次初始化栈 $kpStack$ 和关键前驱变量 $keyPredecessor$（第 1～2 行）；其次，将 $keyPredecessor$ 中的节点入栈，并利用 RPT 表记录的参数的最优提供者信息迭代求取当前节点的关键前驱，当 $keyPredecessor$ 中的节点为 *Source* 时，终止迭代（第 3～6 行）；最后，依次弹出 $kpStack$ 中的节点生成关键路径（第 7 行）.

算法 2.　Backward-KP 过程.

输入：节点可达信息、节点输入的最优提供者

输出：关键路径

```
1. 初始化栈 kpStack;
2. keyPredecessor⇐Destination;
3. WHILE keyPredecessor≠Source DO
4.   kpStack.add(keyPredecessor);
5.   keyPredecessor⇐π(keyPredecessor);
6. ENDWHILE
7. keyPath⇐keyPredecessor+kpStack.popall();
```

例 6.　以图 3 为例说明上述执行过程，见表 6.

① $popBest()$ 是优先队列的元素弹出操作. 在实验中，优先队列是根据 Fibonacci heap[24] 实现的.

表 6 Backward-KP 过程执行步骤

步骤	*keyPredecessor*	*kpStack*
1	*Destination*	
2	m_4	(*Destination*)
3	m_2	(m_4) (*Destination*)
4	m_1	(m_2) (m_4) (*Destination*)
5	*Source*	(m_1) (m_2) (m_4) (*Destination*)
6	弹栈生成关键路径： 〈*Source*，m_1，m_2，m_4，*Destination*〉(*GQoS*=9.9)	

5.2 DAG 解构建(步 3)

关键路径用于指导构建 DAG(当解为链状形式时，此步骤将被省略)．具体过程如下，从 *Destination* 到 *Source* 发起一次沿关键路径的后向搜索生成 DAG：对于关键路径上的节点，其关键输入参数的提供者为关键前驱；对于其他输入参数，选取不改变当前节点关键前驱的提供者．对于非关键路径上的节点，根据 RPT 存储的信息查找其输入参数的最优提供者．需要指出，按照上述过程构建的 DAG，其 GQoS 与关键路径的 *GQoS* 相同．

例 7． 以图 3 为例说明 DAG 的构建过程，见表 7．

表 7 DAG 构建过程

步骤	节点	参数(前驱)
1	*Destination*	$E(m_4)$
2	m_4	$C(m_2)$；$D(m_3)$
3	m_2	$B(m_1)$
4	m_3	A(*Source*)
5	m_1	A(*Source*)
6	*DAG*：*Source*→[(m_1→m_2)‖m_3]→m_4→*Destination*(*GQoS*=9.9)	

5.3 关键路径松弛(KPL)算法(步 4)

如果构建的 DAG 的数量小于阈值 K 时，本文将对最优关键路径进行松弛操作来生成服务质量较差的其他关键路径．这些新的关键路径将继续指导构建与其对应的 DAG．本文将被松弛的关键路径称为最优关键路径，通过松弛操作生成的路径称为非最优关键路径．

定义 6． 松弛[①]操作 Loose．松弛操作的作用对象是关键路径上的节点，它旨在改变当前关键路径上某个节点的关键前驱，以获取该节点新的关键前驱，从而令新关键前驱的 *GQoS* 仅大于旧关键前驱的 *GQoS*．形式化定义如下：对于关键路径 $p=\langle m_i, m_{i+1}, \cdots, m_j\rangle$，$\forall m_k \in p$，其关键前驱 $m_{kp}=m_{k-1}$，$k>i$．给定 m_k 的可达前驱集合 P_{m_k}，$\forall m \in P_{m_k}$ 有：

(1) $m.status=enabled$；

(2) $m.o \in m_k.I$；

令 $m'_{kp}=Loose(m_k)$，则 M'_{kp} 满足：

(3) $m'_{kp} \in P_{m_k}-m_{kp}$；

(4) $m'_{kp}.GQoS>m_{kp}.GQoS$，$k>i$；

(5) $m'_{kp}.GQoS=Min\{m.GQoS \mid m \in P_{m_i}-m_{kp}\}$．

例 8． 以图 10 为例解释松弛操作．选取 m_4 作为松弛节点．由图 10 可知，m_4 输入参数的提供者包括 m_3、m_2 和 m_5，这些节点的 *GQoS* 分别为 7.2、8.2 和 15．其中，m_4 的关键前驱为 m_2．根据松弛操作的定义可得 $m_5=Loose(m_4)$．

非最优关键路径生成：按照从 *Destination* 节点到 *Source* 节点的顺序，依次对关键路径的节点 m_k 进行松弛操作．若存在 $m'_{kp}=Loose(m_k)$，则首先由 m'_{kp} 节点开始执行 Backward-KP 过程求取一条从 *Source* 到 m'_{kp} 的路径，然后将这条路径与最优关键路径包含的从 m'_{kp} 到 *Destination* 的子路径进行组合，得到新的关键路径．需要指出，新求取的关键路径的 *GQoS* 将差于(大于)原有关键路径的 *GQoS*．

例 9． 以图 10 为例，其最优关键路径为〈*Source*，m_1，m_2，m_4，*Destination*〉(*GQoS*=9.9)．图中的虚线双箭头标明了一条非最优关键路径，其具体生成过程见表 8．

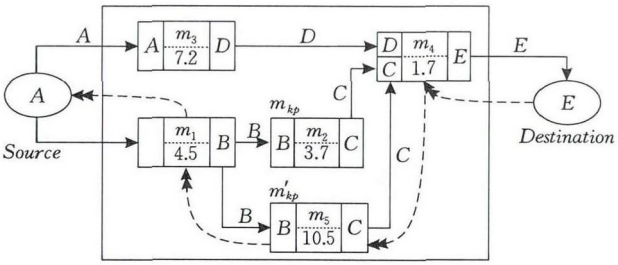


图 10 WAG 示意图

表 8 非最优关键路径生成

步骤	被松弛节点(*GQoS*)	松弛结果(*GQoS*)
1	*Destination*	无
2	m_4(8.2)	m_5(15)
3	m_2	无
4	m_1	无
5	从 m_5 开始执行 Backward-KP 过程得到路径〈*Source*，m_1，m_5〉	
6	〈*Source*，m_1，m_5〉+〈m_4，*Destination*〉生成〈*Source*，m_1，m_5，m_4，*Destination*〉(*GQoS*=16.7)	

本文结合松弛操作实现了关键路径松弛 KPL (Key-Path based Loose)算法．当使用最优关键路径

① 本文之所以使用“松驰”这个动词表示其他非最优主路径的生成操作，是因为“松弛”可以形象地表示出“最优 *GQoS*”的限制被放松来涵盖其他具有较差 *GQoS* 的结果．

不足以生成足够数量的 DAG 时，KPL 算法能够找出定义 2 中满足查询 R 的 Top-K 子图. 算法的具体过程见算法 3：首先，分别使用最优关键路径和数值 k 初始化优先队列 $keyPaths$ 和阈值变量 $threshold$（第 1～2 行）. $keyPaths$ 的性质是具有最小 $GQoS$ 的关键路径将优先弹出. 其次，对 $keyPaths$ 进行循环操作. 当 $keyPaths$ 不为空时，弹出当前最优的关键路径并构建该关键路径对应的 DAG（第 4～5 行）. 如果 DAG 的数量满足阈值要求，则算法结束；否则，更新阈值，并对当前关键路径的节点进行松弛操作. 同时，联合 Backward-KP 过程生成新的 $GQoS$ 较差的关键路径，并将这些关键路径存储到 $keyPaths$ 中（第 7～11 行）.

算法 3.　KPL 算法.

输入：最优关键路径 P，阈值 k

输出：Top-K 子图

```
1.  keyPaths⇐P;
2.  threshold⇐k;
3.  WHILE keyPaths≠∅ DO
4.    keyPath⇐keyPaths.popBest();
5.    DAGs⇐keyPath.constructDAGs();
6.    IF |DAGs|≥threshold THEN return;
7.    threshold⇐threshold−|DAGs|;
8.    FOREACH node v∈keyPath DO
9.      newKeyPath⇐Loose(v)⊕Backward-KP();
10.     keyPaths.add(newKeyPath);
11.   END FOR
12. ENDWHILE
```

6　实　验

6.1　实验设置

基于 Top-K DAG 搜索算法，本文实现了 APISynth 系统来自动地进行对象实例化推荐. 实验设置介绍如下：

实验环境. 2.4 GHz CPU，4 GB RAM，Windows 7 操作系统.

对比方法. 基于全局图模型的方法 Prospector①[2]、基于局部图模型的方法② ParseWeb[5] 和 GraPacc[6].

测试数据集. 公平起见，实验基于对比工作使用的类库和必要的 Java 依赖包构建图模型. 具体而言，与 Prospector 和 ParseWeb 对比时使用了 GEF (Graphical Editing Framework) 类库，而与 GraPacc 对比时使用了 Java SDK Utility (java.util, java.io) 类库. 此外，基于 GEF 类库的 Logic③ 项目源码将用于统计相关 API 的使用频率.

测试查询. 实验使用了 50 个 ($Source$, $Destination$) 形式的查询进行对比实验. 其中包括选自文献[2,5]的 42 个已有查询，以及本文为实验设计的 8 个查询. 需要指出，满足实验所用查询均需要完成较为复杂的编程任务：需要调用构造器函数或执行向上、向下转型操作或调用静态方法.

度量准则. 针对一个查询任务，API 推荐系统将生成多个候选结果供开发人员选择. 为了对相关推荐系统进行评估，本文采用推荐系统常用的评价指标，如查全率、准确率和 F 值来对推荐结果的优劣进行统一衡量. 因为候选结果中排名第一的结果的重要程度最高，所以本文采用信息检索领域广泛使用的度量指标 $P@1$ 评估准确率. 具体而言，查全率 ($Recall$) 表示可被满足的查询在所有查询中的比例. 准确率 ($P@1$) 表示在可被满足的查询中，候选结果列表中 Top 1 结果为正确的查询的比例. F 值表示准确率和查全率的调和平均：$F\text{-score} = 2/(1/P@1+1/Recall)$.

6.2　对比实验

6.2.1　APISynth vs. Prospector

基于文献[2,5]中的查询，本文对比了 APISynth 和 Prospector 的推荐结果.

实验结果见表 9、表 10、表 11 和表 12 中的相关列：Prospector 能够支持的查询数量分别为 18，7，6 和 8，而 APISynth 则分别为 20，11，10④ 和 8. 实验结果表明在能够支持的查询方面，APISynth 优于 Prospector. 原因分析如下：实验中，对于需要 DAG 形式解的查询，Prospector 会返回大量无关的结果，其正确的结果往往被淹没在海量的无效解中. 与 Prospector 相比，APISynth 能够直接搜索出 DAG 形式的解，能对其进行全面、准确评价，因此提升了系统的准确率.

需要特别指出，APISynth 唯一未能解决的查询任务是表 9 中的 ($AbstractDecoratedTextEditor$, $ProjectViewer$). 根据分析，本文作者发现 GEF 类库中不包含 $ProjectViewer$ 类型，因此该查询任务也不能被其他方法解决.

① Prospector 已经演化为 JavaSketch.
② 由于无法获得 XSnippet[3]，本文未对 XSnippet 进行评估.
③ http://www.eclipse.org/downloads/download.php?file=/tools/gef/downloads/drops/R-3.2.1-200609211617/GEF-ALL-3.2.1.zip
④ 每个方法支持查询的数目为 $rank$ 值不为 Nil 的查询数目.

表 9 基于文献[2]中使用查询任务的实验对比结果

查询		Rank	
Source	*Destination*	PROS	APIS
InputStream	*BufferedReader*	1	1
String	*mappedByteBuffer*	1	1
TableViewer	*Table*	1	1
IWorkbench	*IEditorPart*	1	1
ScrollingGraphicalViewer	*FigureCanvas*	1	1
KeyEvent	*Shell*	1	1
Enumeration	*Iterator*	1	1
SelectionChangedEvent	*ISelection*	1	1
ImageRegistry	*ImageDescriptor*	1	1
Map	*Iterator*	1	1
IViewPart	*menumanager*	1	1
TableViewer	*TableColumn*	2	1
IEditorSite	*ISelectionService*	2	1
String	*BufferedReader*	3	1
IWorkbenchPage	*IStructuredSelection*	3	1
IWorkbenchPage	*IDocumentProvider*	3	1
IFile	*String*	4	1
IWorkbenchWindow	*IViewPart*	4	1
AbstractGraphicalEditPart	*ConnectionLayer*	Nil	1
IWorkspace	*IFile*	Nil	2

注：PROS：Prospector，APIS：APISynth，
Nil：当前候选结果集中被评判为正确结果的个数为 0.

表 10 基于文献[5]中使用查询任务的实验对比结果

查询		PARS	PROS	APIS
Source	*Destination*			
ISelection	*ICompilationUnit*	Yes	Yes	Yes
IStructuredSelection	*ICompilationUnit*	Yes	Yes	Yes
ElementChangedEvent	*ICompilationUnit*	Yes	Yes	Yes
IEditorPart	*ICompilationUnit*	Yes	Yes	Yes
IEditorPart	*IEditorInput*	Yes	Yes	Yes
ViewPart	*ISelectionService*	Yes	Yes	Yes
TextEditorAction	*ITextEditor*	Yes	No	Yes
TextEditorAction	*ITextSelection*	Yes	No	Yes
ITextEditor	*ITextSelection*	Yes	Yes	Yes
AbstractDecoratedTextEditor	*ProjectViewer*	No	No	No
ITextEditor	*IDocument*	Yes	No	Yes
ITextEditor	*ITextSelection*	Yes	Yes	Yes

注：PARS：ParseWeb，PROS：Prospector，APIS：APISynth.

表 11 基于 Logic 项目相关查询任务[5]的实验对比结果

查询		PARS		PROS		APIS	
Source	*Destination*	NO	RA	NO	RA	NO	RA
IPageSite	*IActionBars*	1	1	3	1	12	1
ActionRegistry	*IAction*	3	1	4	1	12	2
ActionRegistryProvider	*Contextmenu*	Nil	Nil	2	2	12	1
IPageSiteProvider	*ISelection*	1	1	12	1	12	1
IPageSitemanager	*IToolBar*	2	1	12	1	12	1
String	*ImageDescriptor*	10	6	12	Nil	12	1
Composite	*Control*	10	2	12	Nil	12	3
Composite	*Canvas*	10	5	12	Nil	12	2
GraphicalViewerThumbnail	*Scrollable*	2	1	12	8	12	3
GraphicalViewer	*IFigure*	1	Nil	12	Nil	12	10

注：NO：推荐结果集的大小，RA：用户期待结果的排名，
PARS：ParseWeb，PROS：Prospector，APIS：APISynth，
Nil：当前候选结果集中被评判为正确结果的个数为 0.

表 12 基于 Java SDK Utility 相关查询任务的对比结果

查询		GraP		PROS		APIS	
Source	*Destination*	NO	RA	NO	RA	NO	RA
FileReader	*BufferedReader*	16	1	6	1	12	1
InputStream	*BufferedReader*	16	Nil	10	4	12	1
LinkedHashMap	*Iterator*	36	Nil	12	1	12	9
ArrayList	*Iterator*	26	1	12	1	12	1
FileWriter	*PrintWriter*	3	Nil	4	2	12	1
FileInputStream	*DataInputStream*	36	Nil	20	1	12	1
File	*Scanner*	29	1	12	3	12	1
Pattern	*Matcher*	10	1	6	1	12	1

注：NO：推荐结果集的大小，RA：用户期待结果的排名，
GraP：GraPacc，PROS：Prospector，APIS：APISynth，
Nil：当前候选结果集中被评判为正确结果的个数为 0.

6.2.2 APISynth vs. ParseWeb

基于文献[5]中的查询本文对 APISynth 和 ParseWeb 进行了对比实验. 实验结果见表 10 和表 11 的相关列：ParseWeb 支持的查询数量分别为 11 和 8，而 APISynth 则分别为 11 和 10. 实验结果表明，在支持的查询数量方面 APISynth 优于 ParseWeb. 原因分析如下：由于 ParseWeb 使用语句间的控制流构建图模型，无法对更为精细的 API 之间的数据依赖关系进行刻画，将造成某些查询请求无法被满足. 下面以图 11 为例对此进行解释. 当编程人员输入查询(*GraphicalViewer*，*IFigure*)时，尽管示例代码包含满足查询的方法调用序列，然而 ParseWeb 不能根据这段代码推荐出相关的解. 原因在于 ParseWeb 无法分析出方法 *getGraphicalViewer*()返回值为 *GraphicalViewer* 这一数据依赖关系，从而无法匹配查询包含的类型.

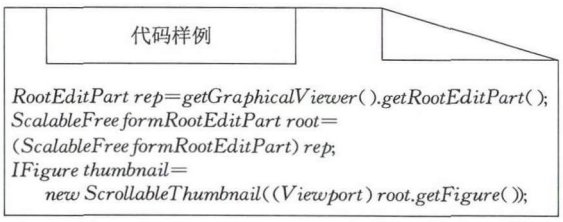


图 11 示例代码

6.2.3 APISynth vs. GraPacc

本小节实验对比了 APISynth 和 GraPacc[6]. 实验结果见表 12：APISynth 能够支持所有的查询，而 GraPacc 仅支持 4 个查询. 实验结果表明，在支持的查询数量方面，APISynth 优于 GraPacc. 原因分析如下：因为 GraPacc 不能利用跨模式的 API 依赖，所以减少了推荐出正确解的可能性. 此外，GraPacc 还受限于模式库中模式的数量. 例如，因为满足查询(*InputStream*，*BufferedReader*)的方法调用序列未包含在模式库中，所以 GraPacc 不支持该查询.

6.3　评价与分析

根据上述对比实验的结果，本文从 3 个方面对相关工作进行评估，包括推荐结果的正确性、最优结果的检测次数以及推荐系统的响应时间.

6.3.1　正确性

根据对比实验结果，本文对相关工作的准确率、查全率和 F 值这些反映结果正确性的指标进行了评估. 需要指出，在被评估的相关工作中，APISynth 和 Prospector 是基于全局图模型的方法，而 ParseWeb 和 GraPacc 是基于局部图模型的方法.

本文采用专家组评判的方法对推荐结果的正确性进行评判. 专家组成员包括 7 名具备 3 年以上 Java 编程经验的程序员. 具体而言，多数专家支持的推荐方案将最终被评判为正确结果. 本文将记录候选结果集中第 1 个被评判为正确结果的位置，如表 9、表 11 和表 12 的 *Rank* 列所示. 当候选结果集中没有任何结果被评判为正确时，相应的 *rank* 值将置为 Nil.

表 13 给出了准确率、查全率和 F 值的评估结果. 由此可知，与基于局部图模型的方法相比，APISynth 将系统的查全率平均提升了 40%；而与基于全局图模型的方法相比，APISynth 将系统的 $P@1$ 提升了 34%. 此外，与其他方法相比，APISynth 将 F 值平均提升了 21%. 这表明 APISynth 能够更好地兼顾查全率和准确率.

表 13　实验结果评价

	查询集 1(见表 9)			查询集 2(见表 11)			查询集 3(见表 12)		
	APISynth	Prospector	改善率	APISynth	ParseWeb	改善率	APISynth	GraPacc	改善率
查全率	1.00	0.90	+10%	1.00	0.80	+20%	1.00	0.40	+60%
$P@1$	0.95	0.61	+34%	0.50	0.56	−6%	0.90	1.00	−10%
F 值	0.97	0.73	+24%	0.67	0.66	+1%	0.95	0.57	+38%
Check#	1.00	1.78	−0.78	1.75	2.25	−0.50	1.00	1.00	0

6.3.2　结果检测次数

衡量 API 推荐系统的一个重要指标是系统能否提高开发人员的编程效率. 为此，依据对比实验结果中的 *Rank* 值，本文统计了用户在找到期望结果之前，需要检测候选结果的平均数目. 例如，若期望结果的 *Rank* 值为 k，则表明找到该结果之前需要用户从头开始查阅排名靠前的 $k-1$ 个结果，于是共检测结果数目为 k 次. 因此，实验计算了相关表中的 *rank* 列值(表 9、表 11 和表 12)的平均值作为结果检测次数，如表 13 中的 *Check*# 所示. 公平起见，实验只选取对比方法双方均可支持的查询任务作为统计源. 实验结果表明，与其他方法相比，使用 APISynth 可显著减少正确结果的检测次数，因此能提高开发人员的编程效率.

6.3.3　系统响应时间

APISynth 借助特殊的数据存储结构能极大地提升算法的执行效率. 实验结果表明，APISynth 完成一次查询的平均时间通常在 200 ms 以内，足以满足开发人员在系统响应速率方面的要求.

需要指出，虽然相比邻接表或邻接矩阵等传统图存储结构，本文采用的存储结构需要耗费更多的内存空间，但是实验结果表明，这对于当今拥有 GB 级别内存的主流计算机不构成挑战性问题. 本文对 APISynth 运行时占用的内存进行了跟踪统计. 结果显示，APISynth 工作时平均只消耗 15 MB 的内存来存储 GEF 类库及其依赖库. 据统计，GEF 类库及其依赖库包含的方法和对象数目均在 60 000 以上，而常用的 44 个类库的方法和对象数目均小于 5000[6,11]. 综上可知，APISynth 在多数情况下可以高效地工作，而不会受到内存容量的限制.

7　结　论

针对对象实例化，本文设计并实现了一种基于新型加权API图(WAG)的API推荐系统——APISynth. 新型 WAG 图可更全面、准确地描述 API 的调用依赖关系，有效地提升了推荐系统的查全率. 此外，本文提出的基于 WAG 图的 Top-K DAG 搜索算法可直接搜索 DAG 形式的解使得系统能对其进行准确地评分. 基于真实类库数据集进行的对比实验表明，与现有方法 Prospector、ParseWeb 和 GraPacc 相比，APISynth 在准确率和查全率两方面获得了更好的表现. 同时 APISynth 的响应时间在 200 ms 以内，能较好地满足开发人员的需求.

在未来的工作中，我们将主要从以下两个方面进一步加强研究：(1) 根据实际需要，在搜索算法中引入更多维度的 QoS，深入研究多维 QoS 场景下最优方案的求取机制；(2) 当类库版本变更时，由于类库中 API 的变化，原有支持旧版本类库的推荐结果常在新版本类库环境下发生编译错误. 为此，我们将研究如何根据新旧版本之间的差异实现过时推荐结果的主动发现和自动迁移.

参 考 文 献

[1] Frakes W, Kang K. Software reuse research: Status and future. IEEE Transactions on Software Engineering, 2005, 31(7): 529-536

[2] Mandelin D, Xu L, Bodlł R, Kimelman D. Jungloid mining: Helping to navigate the API jungle. ACM Sigplan Notices, 2005, 40(6): 48-61

[3] Sahavechaphan N, Claypool K. XSnippet: Mining for sample code. ACM Sigplan Notices, 2006, 41(10): 413-430

[4] Ko A, Myers B, Coblenz M, Aung H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering, 2006, 32(12): 971-987

[5] Thummalapenta S, Xie T. ParseWeb: A programmer assistant for reusing open source code on the web//Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'07). Atlanta, USA, 2007: 204-213

[6] Nguyen A, Nguyen H, Nguyen T T. GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool//Proceedings of the 34th International Conference on Software Engineering (ICSE'12). Zurich, Switzerland, 2012: 1407-1410

[7] Zhong Hao, Zhang Lu, Mei Hong. Mining invocation specifications for API libraries. Journal of Software, 2011, 22(3): 408-416(in Chinese)
(钟浩, 张路, 梅宏. 软件库调用规约挖掘. 软件学报, 2011, 22(3): 408-416)

[8] Yan Y, Xu B, Gu Z, Luo S. A QoS-driven approach for semantic service composition//Proceedings of the 11th IEEE Conference on Commerce and Enterprise Computing(CEC'09). Vienna, Austria, 2009: 523-526

[9] Milanovic N, Malek M. Search strategies for automatic web service composition. International Journal of Web Services Research, 2006, 3(2): 1-32

[10] Jiang W, Zhang C, Huang Z, et al. QSynth: A tool for QoS-aware automatic service composition//Proceedings of the 8th IEEE International Conference on Web Service (ICWS'10). Miami, USA, 2010: 42-49

[11] Zhang C, Yang J, Zhang Y, et al. Automatic parameter recommendation for practical API usage//Proceedings of the 34th International Conference on Software Engineering (ICSE'12). Zurich, Switzerland, 2012: 826-836

[12] Ye Y. Supporting Component-Based Software Development with Active Component Repository Systems[Ph. D. dissertation]. University of Colorado, USA, 2001

[13] Holmes R, Murphy G. Using structural context to recommend source code examples//Proceedings of the 27th International Conference on Software Engineering (ICSE'05). St. Louis, USA, 2005: 117-125

[14] Xie T, Pei J. MAPO: Mining API usages from open source repositories//Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR'06). Shanghai, China, 2006: 54-57

[15] Michail A. Data mining library reuse patterns using generalized association rules//Proceedings of the 22nd International Conference on Software Engineering (ICSE'00). Limerick Ireland, 2000: 167-176

[16] Bruch M, Schäfer T, Mezini M. FrUiT: IDE support for framework understanding//Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange (Eclipse'06). Portland, USA, 2006: 55-59

[17] Saul Z, Filkov V, Devanbu P, Bird C. Recommending random walks//Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07). Luxembourg City, Luxembourg, 2007: 15-24

[18] Uddin G, Dagenais B, Robillard M. Analyzing temporal API usage patterns//Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11). Lawrence, USA, 2011: 456-459

[19] Hummel O, Janjic W, Atkinson C. Code conjurer: Pulling reusable software out of thin air. IEEE Software, 2008, 25(5): 45-52

[20] Kim J, Lee S, Hwang S, Kim S. Towards an intelligent code search engine//Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI'10). Atlanta, USA, 2010: 1358-1363

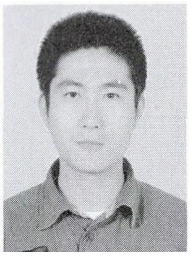
[21] Yu T, Zhang Y, Lin K. Efficient algorithms for Web services selection with end-to-end QoS constraints. ACM Transactions on the Web, 2007, 1(1): 1-26

[22] Zeng L, Benatallah B. QoS-aware middleware for web services composition. IEEE Transactions on Software Engineering, 2004, 30(5): 311-327

[23] Mei Hong, Wang Qian-Xiang, Zhang Lu, Wang Ji. Software analysis: A road map. Chinese Journal of Computers, 2009, 32(9): 1697-1710(in Chinese)
(梅宏, 王千祥, 张路, 王戟. 软件分析技术进展. 计算机学报, 2009, 32(9): 1697-1710)

[24] Fagin R. Combining fuzzy information from multiple systems//Proceedings of the 15th ACM SIGACT-SIGmOD-SIGART Symposium on Principles of Database Systems (PODS'96). Montreal, Canada, 1996: 216-226

[25] Fredman M L, Tarjan R E. Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM, 1987, 34(3): 596-615



LV Chen, born in 1983, Ph. D. candidate. His research interests include software engineering, API usability, and service computing.

JIANG Wei, born in 1985, Ph. D. His research interests include software engineering, API usability, and service computing.

HU Song-Lin, born in 1973, Ph. D., associate professor. His research interests include large scale distributed system and middleware, service computing, and event computing.

Background

Nowadays, increasing attention has been drawn towards the problem of object instantiation in the context of software reuse. However, conventional API recommender systems either cannot guarantee correct output of Direct Acyclic Graph (DAG)-like results or cause the missing of some potential invocation relationships, thereby cannot achieve both good recall and good precision at the same time. In this paper, we propose a tool, called APISynth, to help the developers to perform object instantiation task. A new connected graph is designed to maintain good recall; a new element called "tag" is added to classical graph together with a new reach ability property to avoid false invocation of APIs, and thus enables correct searching of DAG-like results. To achieve good precision, it models the Method Invocation Sequences (MISs) searching problem as a novel Top-*K* DAGs problem. Correspondingly, a Key-Path based Loose (KPL) algorithm is proposed to solve this problem. Therefore, APISynth can significantly improve the recommendation results by increasing both recall and precision.

This work presented in this paper was supported by the National Natural Science Foundation of China under Grant No. 61070027, the State Key Laboratory of Software Engineering (SKLSE2012-09-02). These projects aim to perform some basic research on software engineering. Our group has working in the area of library reuse and software engineering for several years and published many papers. The papers most related to this work, which focus on the recommendation systems, such as API synthesis and service composition, have been published at international conferences, such as ICSE, ICWS, and in academic journals, such as *IEEE Transactions on Services Computing*, *Journal of Computer Science and Technology*, etc.