Compacting Points-To Sets through Object Clustering论文报告

20331021李应东 20331011范兆基

一. 论文概述

针对指针分析，本论文提出了新的指向集表示方式核位向量，减少了表示指向集的内存消耗。该论文还将压缩指向集建模成一个整数规划问题，并利用层级聚类、基于区域聚类和对齐word的标识符映射进以减少指针分析过程所需的空间与时间。

二. 论文详解

1. 指向集->位向量

(1) 连续和稀疏的位向量

假设有n个对象，分别记为1，2，...，n。如果一个连续位向量要表示指向i（0 < i < n）个对象的集合。在最好的情况下，words利用率最大时，即一个word所有位被1填满才用到下一个word，这时候只需$\lceil \frac{i}{W} \rceil$个word，其中每个word有W位。最坏情况下，则需要$\lceil \frac{n}{W} \rceil$

具体的例子：假如有10000个对象，假设为$o_0, o_1, ..., o_{9999}$，有两个指针p和q，它们的指向集分别为

$$pt(p) = o_1, o_{4500}, o_{9999}$$

$$pt(q) = o_1, o_4, o_8$$

那么用位向量来表示就是

$$pt(p) = [\langle 01_1 00\rangle, \langle 0000\rangle, ..., \langle 0000\rangle, \langle 1_{4500} 000\rangle, \langle 0000\rangle, ..., \langle 0000\rangle, \langle 0001_{9999}\rangle]$$

$$pt(q) = [\langle 01_1 00\rangle, \langle 1_4 000\rangle, \langle 1_8 000\rangle]$$

此时word大小W为4，pt(p)达到最坏的情况使用了$\lceil \frac{n}{W} \rceil = \lceil \frac{10000}{4} \rceil$个word，最好的情况时只用$\lceil \frac{i}{W} \rceil = \lceil \frac{3}{4} \rceil$，即一个word就可以表示。而且此时pt(q)为了和pt(p)对齐以方便位运算的操作，需要在末尾填充0达到一样的尺寸，这无疑是很浪费的。

稀疏位向量则表示成如下：

$$pt(p) = \{0\langle 01_1 00\rangle\} \rightarrow \{4500\langle 1_{4500} 000\rangle\} \rightarrow \{9996\langle 0001_{9999}\rangle\} \rightarrow nil$$

$$pt(q) = \{0\langle 01_1 00\rangle\} \rightarrow \{4\langle 1_4 000\rangle\} \rightarrow \{8\langle 1_8 000\rangle\} \rightarrow nil$$

此时多了一些偏移量的数据需要存储 ，减少了很多多余的0，pt(p)得到了极大的改善，pt(q)反而不如之前。当然，如果要满足$pt(p) \subseteq pt(q)$的约束，之前pt(q)需要填充零，而此时不需要，所以pt(q)实际上也是有所改善。

实际上，稀疏位向量最大的的问题并非偏移量数据占用内存，而是向量化
、空间局部性的损失和额外的操作逻辑。

例如对于或操作中，连续位向量只需如下操作：

$$\begin{array}{rl} & [\quad w_1, \quad w_2, \quad \ldots, \quad w_n \quad] \\ | & [\quad w'_1, \quad w'_2, \quad \ldots, \quad w'_n \quad] \\ \hline & [\quad w_1 \mid w'_1, \quad w_2 \mid w'_2, \quad \ldots, \quad w_n \mid w'_n \quad]. \end{array}$$

而稀疏位向量则需要复杂的算法实现：

Algorithm 1: Union of two sparse bit-vectors.

```
Input  : s1, s2 − sparse bit-vectors (linked lists of
                  {offset, word, next} structures).
Output : u − union of s1 and s2.
1  c1 ← s1; c2 ← s2;
2  while c1 ≠ nil ∧ c2 ≠ nil do
3  |  if c1.offset = c2.offset then
4  |  |  u.append(new { c1.offset ⟨c1.word | c2.word⟩ });
5  |  |  c1 ← c1.next; c2 ← c2.next;
6  |  else if c1.offset < c2.offset then
7  |  |  u.append(new { c1.offset ⟨c1.word⟩ });
8  |  |  c1 ← c1.next;
9  |  else
10 |  |  u.append(new { c2.offset ⟨c2.word⟩ });
11 |  |  c2 ← c2.next;
12 |  end
13 end
   // c1 or c2 may not yet be nil; append remainder.
14 while c1 ≠ nil do
15 |  u.append(new { c1.offset ⟨c1.word⟩ });
16 |  c1 ← c1.next;
17 end
18 while c2 ≠ nil do
19 |  u.append(new { c2.offset ⟨c2.word⟩ });
20 |  c2 ← c2.next;
21 end
```

图二

假设一个理想的映射：

$$o_1 \to 0 \quad o_{4500} \to 1 \quad o_{9999} \to 2 \quad o_4 \to 3 \quad o_8 \to 4$$

pt(p)和pt(q)可以表示如下

$$pt(p) = [\langle 1_0 1_1 1_2 0 \rangle]$$

$$pt(q) = [\langle 1_0 001_3 \rangle, \langle 1_4 000 \rangle]$$

此时满足$pt(p) \subseteq pt(q)$的约束，它们之间执行并操作（对应于word的或运算），得到结果$pt(q) = [\langle 1_0 1_1 1_2 1_3 \rangle, \langle 1_4 000 \rangle]$，此时就是最好的情况。但实际中，很难找到这样理想的映射关系。

(2) 核位向量

核位向量是标准位向量的一种改进形式。
例如W=4时，假设5个对象分别映射到9995，9996，9997，9998，9999
核位向量则表示为：

$$\{9992[\langle 0001_{9995} \rangle, \langle 1_{9996} 1_{9997} 1_{9998} 1_{9999} \rangle]\}$$

如果插入9988，则表示为：

$$\{9988[\langle 1_{9988} 000 \rangle, \langle 0001_{9995} \rangle, \langle 1_{9996} 1_{9997} 1_{9998} 1_{9999} \rangle]\}$$

可以看到它使用了稀疏位向量的偏移量，相比起标准位向量完整的从头到尾写出来，核位向量从第一个非零的word开始，到最后一个非零的word，所以相比起稀疏位向量，它中间依旧可能会有大量的为零的word。

2. 压缩指向集

这部分内容主要是：将压缩指向集问题定义为一个整数规划问题及该问题的解决方法。

(1) 定义整数规划问题

我们无法事先预知一个好的对象->标识符映射，论文中使用辅助分析来预估哪些对象将会在同一个指向集中，这种方法的预估结果是主要分析的结果的超集(over-approximate)。因此，为辅助分析创建映射有助于为主要分析创建一个良好的映射(原文：至少比随机映射要好)。

定义整数规划问题的符号、公式较多，所以这部分需要较大的篇幅，且后续变量都是整数。

符号定义：

已知变量：

$$W：word的bit位数量$$

$$P = \{o_{x_1}, o_{x_2}, o_{x_3}, ..., o_{x_n}\}：辅助分析得到的指向集$$

$$n：指向集P中的对象数目$$

$$w = \lceil \frac{n}{W} \rceil：在最优情况(标识符最大程度不分散)下用位向量表示P所需的最少word数量$$

未知变量：

$$m_{x_i}：对象o_{x_i}所映射到的标识符, 1 \leq i \leq n$$

$$f：m_{x_i}起始偏移值的乘数基数$$

压缩指向集的目标是使映射后的标识符尽可能地靠近以节约空间(位向量更小)与时间(并运算更简单)，论文中使用核位向量作为例子，进行后面的分析。

在最优情况下，P中对象的标识符满足以下约束：

$$\frac{|m_{x_i} - m_{x_j}|}{W} < w, 1 \leq i, j \leq n且最小的m_{x_i}分配为f * W$$

此处将首个标识符对齐W的原因是尽可能避免标识符跨越边界。例：W=4，pt={ o_1, o_2 }。若按照如下分配：o_1->7, o_2->8，此时需要2个word。若进行对齐：o_1->0, o_2->1,此时只需要1个word。两种情况都满足 $\frac{|m_{x_i} - m_{x_j}|}{W} < w$，但是后者更优。

该部分定义的整数规划问题的目的是求出最优的m(标识符)和f(乘数基数)，论文按照渐进的方式得到最终的整数规划问题。

限制C1：只需要指向集中所有对象被分配的标识符大于起始偏移且只需要理想化最小的word数目。

$$m_{x_i} \geq f_p \cdot W$$

$$m_{x_i} < f_p \cdot W + w_p \cdot W$$

$$f_p \geq 0$$

限制C2：不同对象不能映射到同一个标识符。

形式一：

$$m_{x_i} - m_{x_j} > 0$$

形式二：

$$m_{xi} - m_{x_j} < L \cdot b_{ij}$$

$$m_{xi} - m_{xj} > -L + L \cdot b_{ij}$$

$$b_{ij} \geq 0$$

$$b_{ij} \leq 0$$

该形式的约束通过$b_{ij} = 0$与$b_{ij} = 1$的分类讨论可以较好理解

仅仅C1/C2两个约束会出现鸽巢原理的问题：

假设有W个指向集，每个指向集都有一个共同元素o(标识符为m，在w个word钟)，然后每个指向集还有一个独一无二的对象。在这种情况下，总共有(W+1)个对象，一个word(大小为W)无法表示。考虑(W-1)个指向集，则一个word正好可以表示。多出来的第W个指向集的唯一对象最好是分配在第(w-1)或(w+1)个word中。

限制C3：论文中的解决方法是为每个指向集引入容忍值去拓展标识符的范围，同时为每个指向集引入一个未知变量 t, 衡量拓展范围的大小。

$$m_{x_i} \geq f_p \cdot W$$

$$m_{xi} < f_p \cdot W + w_p \cdot + W + t_p \cdot W$$

$$f_p \geq 0$$

$$t \geq 0$$

容易发现 C3 是 C1 的改进版，但是容忍度不可过大，否则就违背了精简指向集的初衷。所以规划问题的另一目标就是最小化 $t_1 + t_2 + t_3 + ...t_n$

总结起来就是：对每个指向集P，在最优情况为使用w_p个word的前提下，将指向集P表示为核位向量。上述限制限定了一个使指向集适用于w_p个word的标识符映射，但是允许对word有一定的容忍度。最小化容忍乘数($t_1 + t_2 + t_3 + ...t_n$)的优化用以保证使用尽可能少的word表示指向集，从而得到理想的标识符映射。

实际上这个整数规划问题的求解十分复杂，只有为数不多的方法可以简化该问题的求解，其中之一就是后续提到的用于优化聚类的分区，将所有对象划分成不同的群组单独考虑。使用整数规划求解指向集压缩问题不能与当前用于大型程序的技术相提并论。后续的内容关注更粗略但适配性更强的方法。

(2) 层级聚类

我们的目标是将有关联的被指向对象（比如出现在同一指向集的）分配相近的数字标识符，通过层级聚类来实现这一目标。本文使用自底向上的聚类方式，即一开始有若干个单对象簇，然后根据距离最近的原则不断聚合成簇，直至只剩一个簇。

例如：

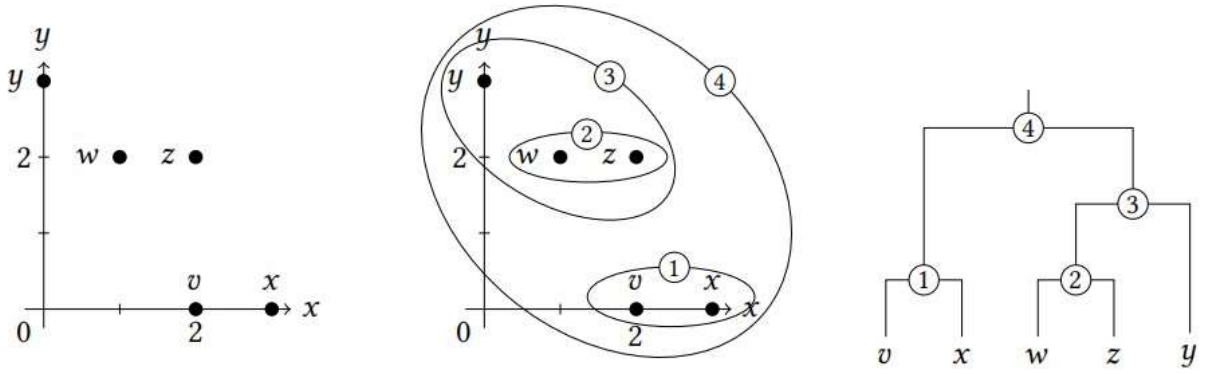

Fig. 3. Agglomerative clustering process (b) of coordinate data (a) and the resulting dendrogram (c).

在这个例子中，根据坐标的欧式距离，按图(b)序号先后聚合，得到最终一个簇④，同时得到图(c)这样的层次结构。　　有三种连接标准来判断是否在簇之间做聚合操作：- 单连接：通过两个簇内的最近的对象间的距离来判断，例如图(b)中聚合3时计算w和y的距离 - 完全连接：同上相反，找最远距离，例如图(b)中聚合3时计算z和y的距离 - 平均连接：$\frac{1}{|C_1||C_2|}\sum\limits_{o_1\in{C_1}}\sum\limits_{o_2\in{C_2}}d(o_1,o_2) $，例如图(c)中聚合3时计算w和z连线中点到y的距离

聚类在本论文的数据上运行得非常快，所以论文在三种标准上都进行测试，最后选出可能得到理想映射的聚类结果。

(3) 聚类对象

想在具体在指向的对象上实现聚类，首先要知道如何定义两个对象之间的距离，就好像上面图三的例子，坐标上的点间的距离可以用欧式距离来衡量。

(论文中)对象间的距离：同时包含这两个对象的指向集的最小word数。

比如，考虑指向集$\{o_a, o_b, o_c, o_d, o_e, o_f\}$和$\{o_a, o_f\}$，如果我们逐一扫描这两个指向集，可能得到这样的映射：

$$o_a \to 0 \quad o_b \to 1 \quad o_c \to 2 \quad o_d \to 3 \quad o_e \to 4 \quad o_f \to 5$$

此时第一个指向集会用两个word来表示，这是理想的，但是后一个指向集也需要两个word来表示，但实际只需要一个word就可以。如果我们根据上述距离的定义，可以得到d(o_a, o_f)=min(2,1)=1，o_a与其他对象的距离为2，o_f与其他对象距离为∞，那么o_a和o_f距离最近，应分配更近的数字标识符，此时的映射可能如下：

$$o_a \to 0 \quad o_f \to 1 \quad o_b \to 2 \quad o_c \to 3 \quad o_d \to 4 \quad o_e \to 5$$

这时第一个指向集的word数仍为2，而后一个的word数则为1了，达到最理想情况。通过这个定义，我们就可以建立一个距离矩阵来进行聚类操作。矩阵元素先初始化为无穷，然后扫描每一个指向集来更新矩阵数据，得到距离矩阵。

有了距离矩阵就可以进行聚类，得到树状图。在树状图上进行深度优先搜索得到的对象就是最邻近的对象。深搜得到的叶结点顺序配合从零开始的计数器分配标识符以实现对象→标识符映射。

(4) 基于区域的聚类

距离矩阵是对称矩阵，所以可以压缩成上三角矩阵(不包含对角线)，但矩阵元素为$\frac{n(n-1)}{2}$，内存会随着元素的增加呈二次增长，如下图：

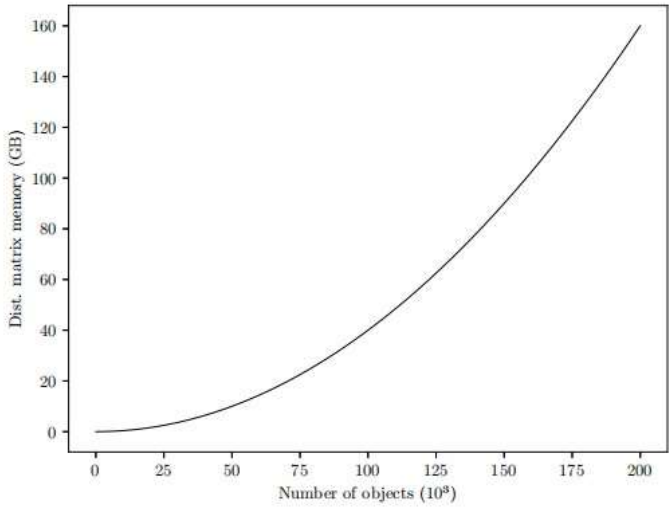

Fig. 4. Memory used to represent various (condensed) distance matrices.

为了减少内存消耗，我们将对象分组到独立的区域，在不牺牲精度的情况下对每个区域进行聚类。能够实现这点是因为许多对象之间没有任何联系，它们从未出现在同一指向集中，所以无论它们的标识符离得多远都不会有不利影响。　　具体做法就是：建立一个无向图，在出现于同一指向集的对象间加上边，同一连通分支划分为一个区域，对每个区域分别进行聚类，此时标识符的计数器为上一区域结束时的标识符。这样多了很多距离矩阵，但这些距离矩阵所占内存之和是比之前一整个距离矩阵的内存要小的，比如n个对象均分为两个区域，此时两个矩阵元素总数为$\frac{\frac{n}{2}(\frac{n}{2}-1)}{2}*2=\frac{n(\frac{n}{2}-1)}{2}$，一个矩阵时则为$\frac{n(n-1)}{2}$，前者更小。此时我们不会对多个区域并发进行聚类，因为每个距离矩阵在完成聚类，分配好标识符后便可以释放，因此顺序执行聚类更节约内存。　　部分区域的对象小于W，该区域只需分配一个word，无需聚类操作来分析它们的邻近关系，直接分配任意标识符即可，文中称这些区域为琐碎区域。

Table 2. Number of memory objects, regions, non-trivial regions, objects in non-trivial objects (with the proportion of total objects they make up), and the number of objects in the largest region for our benchmark programs.

Program	Memory Objects	Regions	Non-Trivial Regions	Non-Trivial Region Objects	Largest Region
dhcpcd	3699	2903	1	569 (15.38%)	569
gawk	4784	3061	1	1138 (23.79%)	1138
bash	4339	3247	2	644 (14.84%)	553
mutt	7186	4265	1	2434 (33.87%)	2434
lynx	7917	4177	3	1949 (24.61%)	1699
xpdf	19 101	14 328	1	3468 (18.16%)	3468
ruby	20 235	9073	25	9331 (46.11%)	5069
keepassxc	37 671	28 468	9	5224 (13.87%)	4026

如图，在本论文的测试样例中，可以看出琐碎区域占了占比较大，这将大大减少了聚类操作的工作负载。

(5) 对齐word的标识符映射

对于每一个区域，该区域内的标识符映射从W(word的大小）的整数倍开始计数。这样可以尽量减少每一个区域进行标识符分配后各指向集所用的word数量。

考虑如下例子(虽然此处以琐碎区域举例，但是结论可以拓展至至非琐碎区域)：有两个rgionR_1(包含o_a, o_b, o_c)和R_2(包含o_d, o_e, o_f)，W = 4

不对齐word的情况：两个区域的标识符分配分别为$R_1 : o_a \to 0, o_b \to 1, o_c \to 2$ 和 $R_2 : o_d \to 3, o_e \to 4, o_f \to 5$。包含$R_1$中对象的指向集可以表示为$\{0[\langle \times \times \times 1 \rangle]\}$，即$\{o_a, o_b, o_c\}$的非空子集，其中×代表0或1。同理，包含$R_2$中对象的指向集具有以下三种形式：

$$\{0[\langle 0001 \rangle]\} <==> \{o_d\}$$

$$\{4[\langle \times \times 01 \rangle]\} <==> \{o_c\}、\{o_d\}、\{o_c, o_d\}$$

$$\{0[\langle 0001\rangle, \langle \times \times 01\rangle]\} <==> \{o_d, o_e\}, \{o_d, o_f\}, \{o_d, o_e, o_f\}$$

对于包含R_2中对象的指向集的第三种形式，原本只需要1个word即可，但是现在却需要2个word。导致这样的原因是：不存在同时包含R_1中对象和R_2中对象的指向集，但这两个区域却共享了word，这样就会存在区域并非是最好地利用word。

对齐的情况：两个区域的标识符分配分别为$R_1: o_a \to 0, o_b \to 1, o_c \to 2$ 和 $R_2: o_d \to 4, o_e \to 5, o_f \to 6$。此时$R_2$的标识符分配按照W (W = 4) 进行了对齐，包含R_2中对象的指向集就只有一种形式$\{4[\langle \times \times \times 11\rangle]\}$，$R_2$只需要一个word即可，各指向集只需要使用理论上最少的word数目，优于不对齐的分配方式。

对齐word的标识符映射会使位向量中留下间隙，比如上述例子中的3号位就没有被使用。但是这个空缺无关紧要，因为指向集只会包含它所属区域内的对象，这个空缺是多出来的，不会被用到。更进一步地说，由于对齐而导致的核向量开头的多余的0也无关紧要。虽然改论文是在辅助分析的结果下进行聚类，但主阶段分析生成的指向集是辅助分析生成的指向集的子类，主阶段分析中的区域不会大于辅助分析中的，所以空缺的不重要性也可以适用于至主阶段分析。

3. 评估

(1) 实现

论文已在SVF框架下使用fastcluster实现了上述方法，其中稀疏位向量用LLVM中的实现，使用STL向量实现核位向量，标准位向量对应核位向量偏移量为零的情况。更多内容看后面代码讲解部分，下面对评估效果略作介绍。

(2) Benchmark

本文使用了8个基准程序做实验，具体信息如下：

Table 3. Benchmark versions, bitcode sizes in MiB, lines of LLVM instructions, and descriptions.

Benchmark	Version	Size	LOI	Description
dhcpcd	9.3.4	1.2 MiB	82 939	Dynamic host configuration protocol client
gawk	5.1.0	2.4 MiB	179 805	GNU AWK; text filtering language interpreter
bash	5.0.18	2.6 MiB	196 168	Bourne Again Shell; Unix shell
mutt	2.0.3	3.2 MiB	224 500	Text-based email client
lynx	2.8.9	5.1 MiB	286 991	Text-based web browser
xpdf	4.03	7.6 MiB	494 764	PDF viewer
ruby	2.7.2	13 MiB	864 114	Ruby programming language interpreter
keepassxc	2.6.2	15 MiB	828 669	Password manager

(3) 实验效果

表四展示了使用稀疏位向量时，分别在理想条件下，原始映射下，和本文方法在三个连接标准下所需的word数，可以看出本文方法不一定总在同一标准下表现最优，但是三个标准下都优于原始情况，而三个标准的最少word数仅在理想条件的1~3倍左右。

Table 4. Number of words required for sparse bit-vectors theoretically, in the original naive mapping, and under mappings produced using various linkage criteria. The reduction compares the result predicted to be best (in bold) against the original.

Benchmark	Theoretical	Original	Single	Complete	Average	Reduction
dhcpcd	3 317 195	24 726 044	**4 991 412**	6 635 746	6 635 082	4.95×
gawk	58 007 460	429 843 096	**82 989 102**	132 528 508	99 502 900	5.18×
bash	26 586 881	289 532 162	**42 914 256**	42 914 700	53 173 774	6.75×
mutt	51 298 142	490 533 016	**102 662 924**	145 767 682	160 026 830	4.78×
lynx	133 664 618	965 029 738	267 599 228	319 144 056	**215 831 960**	4.47×
xpdf	731 879 787	3 943 399 840	1 463 766 422	1 527 354 270	**1 400 196 420**	2.82×
ruby	320 059 196	3 920 937 120	888 289 648	889 035 364	**764 713 778**	5.13×
keepassxc	13 770 856	315 331 268	**47 619 934**	74 384 858	67 691 754	6.62×
Geo. Mean						4.94×

表五则是使用核位向量的情况，可以看到和表四类似的结论。

Table 5. Number of words required for core bit-vectors theoretically, in the original naive mapping, and under mappings produced using various linkage criteria. The reduction compares the result predicted to be best (in bold) against the original.

Benchmark	Theoretical	Original	Single	Complete	Average	Reduction
dhcpcd	3 317 195	23 911 465	**4 961 417**	6 605 816	5 784 023	4.82 ×
gawk	58 007 460	429 739 789	**82 783 110**	140 588 641	148 836 214	5.19 ×
bash	26 586 881	295 168 808	**31 731 607**	36 861 568	47 120 912	9.30 ×
mutt	51 298 142	548 971 273	**87 213 543**	260 457 927	259 746 461	6.29 ×
lynx	133 664 618	1 015 676 964	**237 113 529**	289 849 510	302 122 259	4.28 ×
xpdf	731 879 787	4 197 513 654	1 558 434 196	1 558 496 134	**1 526 729 185**	2.75 ×
ruby	320 059 196	6 600 730 356	**1 405 659 097**	2 514 836 137	2 186 425 117	4.70 ×
keepassxc	13 770 856	1 399 786 369	**107 456 539**	134 257 502	120 881 288	13.03×
Geo. Mean						5.66 ×

表六为使用稀疏位向量(SBV)、标准位向量(BV)和核位向量(CBV)分别在非聚合和聚合情况下运行SFS的时间，单位为秒。OOM和OOT分别表示分析耗尽的可用内存和时间。可以看到相同条件下，使用了本文方法的时间都缩短了，OOM的情况也消失。

Table 6. Time to run SFS using unclustered identifiers with sparse bit-vectors (SBV), bit-vectors (BV), and core bit-vectors (CBV) and using clustered identifiers with SBVs and CBVs, in seconds. Using unclustered identifiers with SBVs and BVs forms the baseline. The best result achieved by the baseline and by our approach are compared. OOM and OOT mean the analysis exhausted available memory and time, respectively.

Benchmark	Unclustered			Clustered		Improvement	
	SBV	BV	CBV	SBV	CBV	SBV/SBV	BV/CBV
dhcpcd	66.50	59.09	58.40	52.93	52.87	1.25×	1.12×
gawk	1417.51	1002.50	971.71	773.34	655.85	1.83×	1.53×
bash	307.88	218.72	228.09	188.46	175.58	1.63×	1.25×
mutt	666.08	517.56	464.38	396.15	360.53	1.68×	1.44×
lynx	3058.57	2531.57	2383.98	1889.04	1880.56	1.62×	1.35×
xpdf	OOM	10 836.70	10 518.00	10 904.40	9636.46	-	1.12×
ruby	OOM	OOM	7451.52	6159.25	5985.72	-	-
keepassxc	1084.50	1093.42	1049.59	624.52	628.30	1.74×	1.74×
Geo. Mean						1.62×	1.35×

表七与表六为同一实验，不过记录的是内存。同样可以看到改进，并且OOM的情况消失。

Table 7. Memory required to run SFS using unclustered identifiers with sparse bit-vectors (SBV), bit-vectors (BV), and core bit-vectors (CBV) and using clustered identifiers with SBVs and CBVs, in gigabytes. Using unclustered identifiers with SBVs and BVs forms the baseline. The best result achieved by the baseline and by our approach are compared. OOM and OOT mean the analysis exhausted available memory and time, respectively.

Benchmark	Unclustered			Clustered		Improvement	
	SBV	BV	CBV	SBV	CBV	SBV/SBV	BV/CBV
dhcpcd	1.20	0.92	0.89	0.74	0.68	1.62×	1.35×
gawk	12.76	8.02	7.76	4.63	3.67	2.75×	2.19×
bash	9.00	4.94	5.06	3.23	2.67	2.79×	1.86×
mutt	14.56	11.68	11.42	5.47	4.56	2.66×	2.56×
lynx	29.07	21.83	19.23	11.51	9.38	2.53×	2.33×
xpdf	OOM	72.08	68.62	42.47	29.91	≥2.35×	2.41×
ruby	OOM	OOM	86.91	32.95	34.67	≥3.04×	≥2.88×
keepassxc	12.41	25.19	24.31	6.15	6.22	2.02×	4.05×
Geo. Mean						≥2.43×	≥2.35×

三. 代码讲解

1. 代码内容

(1) 代码概述

本论文并没有提出一个新的源代码分析框架，只是对指针分析的PointToSet的表示方式进行改进。所以本论文的代码是在SVF框架上进行一定改动，将论文内容嵌入SVF框架中，对指针分析的PointToSet部分进行了一定的改动。论文代码的ReadMe文件中提到：代码的主要改动为三个部分——CoreBitVector、NodeIDAllocator和PointsTo。后面的代码讲解就主要关注这三个部分。

(2) CoreBitVector

这部分代码包含两个代码文件CoreBitVector.h和CoreBitVector.cpp，主要内容是CoreBitVector和CoreBitVectorIterator这两个类，分别实现了CBV及其迭代器。从类的层次结构来看，CoreBitVectorIterator是CoreBitVector的内部类

① CoreBitVector

CoreBitVector类是CBV的代码实现，一个CoreBitVector对象代表一个CBV。该类对CBV的结构实现如下代码所示：

```
class CoreBitVector
{
    public:
    typedef unsigned long long Word;// 一个Word代表64bit

    // ......省略

    private:
    // The first bit of the first word.
    // CBV中第一个有效word(非0word)的第一个有效bit在整个CBV中的偏移量
    unsigned offset;
    // Our actual bit vector.
    // 存储第一个有效word及其之后的word
    std::vector<Word> words;
}
```

此外，该类实现了大部分的集合运算，如：交(&=)、并(|=)、差(-=)等，足以支持SVF框架中的指针分析。

② CoreBitVectorIterator

CoreBitVectorIterator类实现了CBV的迭代器，结构实现的代码如下：

```
class CoreBitVectorIterator
{
    //.......省略

    private:
    /// CoreBitVector we are iterating over.(当前CoreBitVector)
    const CoreBitVector *cbv;
    /// Word in words we are looking at.(当前CoreBitVector中的当前Word)
    std::vector<Word>::const_iterator wordIt;
    /// Current bit in wordIt we are looking at(当前Word中的bit偏移量)
    /// (index into *wordIt).
    unsigned bit;
};
```

该类实现的方法包含了基本的迭代器功能，如：解引用(*)、自增(前置/后置++)、赋值(=)等。使用该迭代器对CoreBitVector对象进行迭代，可以得到CoreBitVector对象所蕴含的CBV信息，即CBV有哪些位置被置1。

(3) NodeIDAllocator

这部分代码包含两个代码文件NodeIDAllocator.h和NodeIDAllocator.cpp，主要内容是NodeIDAllocator和Clusterer这两个类。从类的层次结构来看，Cluster是NodeIDAllocator的内部类。这部分的代码所实现的主要功能是聚类算法中的聚类以及代码运行过程中对各值、对象的标识符分配。

① NodeIDAllocator

这个类实现的是对值、对象的标识符分配，根据策略不同会有不同的分配方式(DENSE、SEQ和DEBUG)，而其中有一类GepObject比较特殊，在DEBUG策略中标识符分配方式会有所变化。

② Cluster

Cluster，顾名思义就是实现了聚类算法的类，但是在该论文提供的代码中，该类只提供在DENCE标识符分配策略下的聚类。这个类实现的重要方法有：regionObjects(根据辅助分析建立的无向图划分有关联的对象)、getDistanceMatrix(得到聚类算法中所需要的距离矩阵，存储各节点(值或对象)间距离)、traverseDendogram(利用深度优先搜索遍历聚类树，依次得到各个叶子结点)、evaluate(评估使用不同类型PonitToSet(原始BV、离散BV等)的效果)、cluster(根据节点标识符进行聚类)、getReverseNodeMapping(反映射cluster()函数的聚类结果)

(4) PointsTo

这部分代码包含两个代码文件PointsTo.h和PointsTo.cpp，包含PointsTo和PointsToIterator两个类(PointsToIterator是PointsTo的内部类)。这部分代码的主要作用就是表示一个PointToSet。总体而言这部分代码并没有太多的新内容，主要是将表示一个PointToSet所需要的内容整合起来。首先，一个PointsTo类需要确定PointToSet的类型(BV/SBV/CBV)，然后根据这个类型选择PointsTO类中用以表示BitVector的对象(暂称bv)及PointsToIterator中的迭代器(暂称It)。PointsTo中实现的各函数基本都是调用bv中对应的函数，相应地，PointsToIterator中的函数基本也是调用It的相应函数。此外，PointsTo类还会存储聚类的映射及反映射。

2. 代码运行

(1) 运行结果

TABLE 4: Required words for SBV

Benchmark	Theoretical	Orginal	Single	Complete	Average	Reduction
dhcpcd.bc	3317195	24726024	*4991412*	6635746	6635082	4.95x
gawk.bc	58007460	429843180	*82989102*	132528508	99502900	5.18x
bash.bc	26586881	289532162	*42914256*	42914700	53173774	6.75x
mutt.bc	51298142	490532984	*102662924*	145767658	160026830	4.78x
lynx.bc	133664618	965029716	267599228	319144056	*215831960*	4.47x
xpdf.bc	--	--	--	--	--	--x
ruby.bc	--	--	--	--	--	--x
keepassxc.bc	13770856	315331336	*54312908*	74407698	74381880	5.81x
					Geo. mean	5.27x

TABLE 5: Required words for CBV

Benchmark	Theoretical	Orginal	Single	Complete	Average	Reduction
dhcpcd.bc	3317195	23911464	*4961417*	6605816	5784023	4.82x
gawk.bc	58007460	429739626	*82783110*	140588641	148836214	5.19x
bash.bc	26586881	295168815	*31731607*	36861568	47120912	9.30x
mutt.bc	51298142	548971337	*87213543*	260457927	259746461	6.29x
lynx.bc	133664618	1015676938	*237113529*	289849510	302122259	4.28x
xpdf.bc	--	--	--	--	--	--x
ruby.bc	--	--	--	--	--	--x
keepassxc.bc	13770856	1399785524	*107456545*	134257494	120881288	13.03x
					Geo. mean	6.59x

TABLE 6: Time

Benchmark	Unclustered			Clustered		SBV/SBV	BV/CBV
	SBV [s]	BV [s]	CBV [s]	SBV [s]	CBV [s]		
dhcpcd.bc	54.85	51.03	63.92	46.13	46.74	1.19x	1.09x
gawk.bc	879.08	732.79	731.83	624.72	601.62	1.41x	1.22x
bash.bc	204.30	165.37	166.57	140.13	140.62	1.46x	1.18x
mutt.bc	496.33	438.60	427.53	370.20	359.54	1.34x	1.22x
lynx.bc	OOM	OOM	OOM	1532.27	1560.07	--x	--x
xpdf.bc	OOM	OOM	OOM	OOM	OOM	--x	--x
ruby.bc	OOM	OOM	OOM	OOM	OOM	--x	--x
keepassxc.bc	688.88	OOM	OOM	515.54	511.04	1.34x	--x
					Geo. mean	1.34x	1.18x

TABLE 7: Memory

Benchmark	Unclustered			Clustered		SBV/SBV	BV/CBV
	SBV [GB]	BV [GB]	CBV [GB]	SBV [GB]	CBV [GB]		
dhcpcd.bc	1.20	0.92	0.91	0.74	0.68	1.62x	1.34x
gawk.bc	12.76	8.00	7.79	4.63	3.67	2.75x	2.18x
bash.bc	9.00	4.93	5.06	3.23	2.66	2.79x	1.85x
mutt.bc	14.28	11.67	11.45	5.47	4.56	2.61x	2.56x
lynx.bc	OOM	OOM	OOM	11.52	9.37	>=1.30x	>=1.60x
xpdf.bc	OOM	OOM	OOM	OOM	OOM	--x	--x
ruby.bc	OOM	OOM	OOM	OOM	OOM	--x	--x
keepassxc.bc	12.41	OOM	OOM	6.30	6.21	1.97x	>=2.41x
					Geo. mean	>=2.09x	>=1.94x

(2)论文结果

Table 4. Number of words required for sparse bit-vectors theoretically, in the original naive mapping, and under mappings produced using various linkage criteria. The reduction compares the result predicted to be best (in bold) against the original.

Benchmark	Theoretical	Original	Single	Complete	Average	Reduction
dhcpcd	3 317 195	24 726 044	**4 991 412**	6 635 746	6 635 082	4.95×
gawk	58 007 460	429 843 096	**82 989 102**	132 528 508	99 502 900	5.18×
bash	26 586 881	289 532 162	**42 914 256**	42 914 700	53 173 774	6.75×
mutt	51 298 142	490 533 016	**102 662 924**	145 767 682	160 026 830	4.78×
lynx	133 664 618	965 029 738	267 599 228	319 144 056	**215 831 960**	4.47×
xpdf	731 879 787	3 943 399 840	1 463 766 422	1 527 354 270	**1 400 196 420**	2.82×
ruby	320 059 196	3 920 937 120	888 289 648	889 035 364	**764 713 778**	5.13×
keepassxc	13 770 856	315 331 268	**47 619 934**	74 384 858	67 691 754	6.62×
Geo. Mean						4.94×

Table 5. Number of words required for core bit-vectors theoretically, in the original naive mapping, and under mappings produced using various linkage criteria. The reduction compares the result predicted to be best (in bold) against the original.

Benchmark	Theoretical	Original	Single	Complete	Average	Reduction
dhcpcd	3 317 195	23 911 465	**4 961 417**	6 605 816	5 784 023	4.82 ×
gawk	58 007 460	429 739 789	**82 783 110**	140 588 641	148 836 214	5.19 ×
bash	26 586 881	295 168 808	**31 731 607**	36 861 568	47 120 912	9.30 ×
mutt	51 298 142	548 971 273	**87 213 543**	260 457 927	259 746 461	6.29 ×
lynx	133 664 618	1 015 676 964	**237 113 529**	289 849 510	302 122 259	4.28 ×
xpdf	731 879 787	4 197 513 654	1 558 434 196	1 558 496 134	**1 526 729 185**	2.75 ×
ruby	320 059 196	6 600 730 356	**1 405 659 097**	2 514 836 137	2 186 425 117	4.70 ×
keepassxc	13 770 856	1 399 786 369	**107 456 539**	134 257 502	120 881 288	13.03×
Geo. Mean						5.66 ×

Table 6. Time to run SFS using unclustered identifiers with sparse bit-vectors (SBV), bit-vectors (BV), and core bit-vectors (CBV) and using clustered identifiers with SBVs and CBVs, in seconds. Using unclustered identifiers with SBVs and BVs forms the baseline. The best result achieved by the baseline and by our approach are compared. OOM and OOT mean the analysis exhausted available memory and time, respectively.

Benchmark	Unclustered			Clustered		Improvement	
	SBV	BV	CBV	SBV	CBV	SBV/SBV	BV/CBV
dhcpcd	66.50	59.09	58.40	52.93	52.87	1.25×	1.12×
gawk	1417.51	1002.50	971.71	773.34	655.85	1.83×	1.53×
bash	307.88	218.72	228.09	188.46	175.58	1.63×	1.25×
mutt	666.08	517.56	464.38	396.15	360.53	1.68×	1.44×
lynx	3058.57	2531.57	2383.98	1889.04	1880.56	1.62×	1.35×
xpdf	OOM	10 836.70	10 518.00	10 904.40	9636.46	-	1.12×
ruby	OOM	OOM	7451.52	6159.25	5985.72	-	-
keepassxc	1084.50	1093.42	1049.59	624.52	628.30	1.74×	1.74×
Geo. Mean						1.62×	1.35×

Table 7. Memory required to run SFS using unclustered identifiers with sparse bit-vectors (SBV), bit-vectors (BV), and core bit-vectors (CBV) and using clustered identifiers with SBVs and CBVs, in gigabytes. Using unclustered identifiers with SBVs and BVs forms the baseline. The best result achieved by the baseline and by our approach are compared. OOM and OOT mean the analysis exhausted available memory and time, respectively.

Benchmark	Unclustered			Clustered		Improvement	
	SBV	BV	CBV	SBV	CBV	SBV/SBV	BV/CBV
dhcpcd	1.20	0.92	0.89	0.74	0.68	1.62×	1.35×
gawk	12.76	8.02	7.76	4.63	3.67	2.75×	2.19×
bash	9.00	4.94	5.06	3.23	2.67	2.79×	1.86×
mutt	14.56	11.68	11.42	5.47	4.56	2.66×	2.56×
lynx	29.07	21.83	19.23	11.51	9.38	2.53×	2.33×
xpdf	OOM	72.08	68.62	42.47	29.91	≥2.35×	2.41×
ruby	OOM	OOM	86.91	32.95	34.67	≥3.04×	≥2.88×
keepassxc	12.41	25.19	24.31	6.15	6.22	2.02×	4.05×
Geo. Mean						≥2.43×	≥2.35×

(3) 结果比对与分析

(1) 相同

比对我们的运行结果与论文给出的实验结果，可以发现在没有发生OOM(内存溢出)和OOT(超时)的情况下，两者的数据基本一致，这说明我们小组使用论文给出的代码可以基本复现出作者的实验结果。

(2) 不同

在部分论文结果有数据的位置，我们的复现结果为"--"或者"OOM"，导致这种情况的原因是运行内存不足。论文的实验环境是100G的运行内存，我们小组在复现时使用的是个人电脑，只能在15G运行内存的环境下进行运行，无法分析论文中给出的一些待分析源码样例。

四. 论文见解

该论文提出了新形式的位向量核位向量及其对应的优化方法。我们小组认为该论文提出的核位向量可以应用于其他很多地方。一些原本使用位向量的地方都可以尝试一下使用该论文的内容，特别是位向量比较稀疏且数量较大的情况，从而减少内存消耗。

五. 成员分工

各小组成员均参与该报告的所有部分且贡献相同。