gSpan算法实验报告

海量图数据的管理和挖掘

2015 年 10 月 31 日

作者：马凌霄

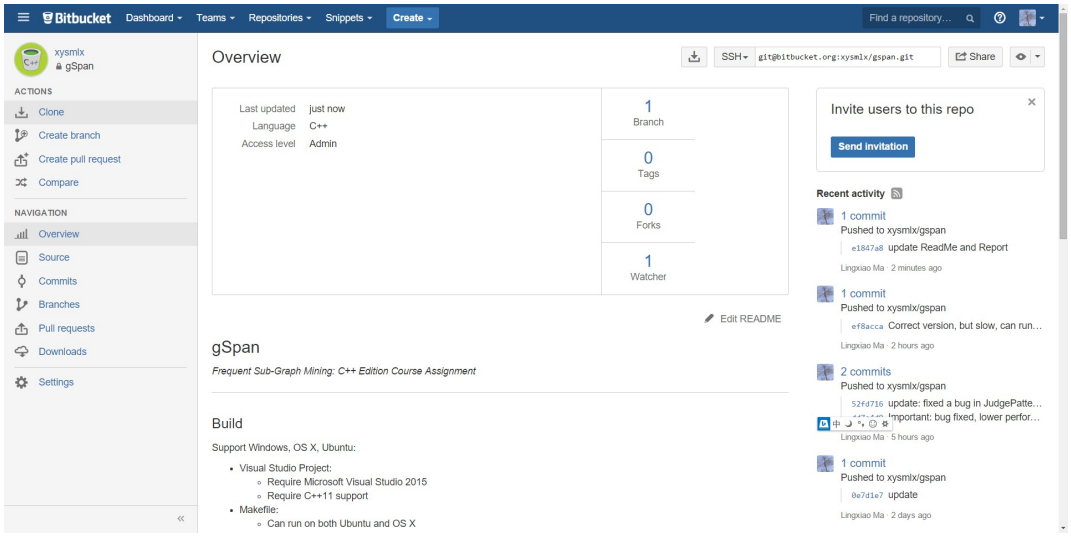
学号：1501111302

院系：信息科学技术学院

EMail：xysmlx@gmail.com

Repository (Private): https://bitbucket.org/xysmlx/gspan

Git (Remote): https://xysmlx@bitbucket.org/xysmlx/gspan.git



目　　录

§ 1 gSpan算法简述

gSpan是Xifeng Yan和Jiawei Han在ICDM 02的论文[1]《gSpan: Graph-Based Substructure Pattern Mining》提出的用于挖掘频繁子图的算法，该论文的主要贡献有两点：

- 提出新的频繁子图挖掘算法，比当时最快的FSG算法快数倍；
- 提出DFSCODE理论，用于解决频繁字图挖掘的核心问题：子图同构问题。

定义1. 频繁子图：给定图集合G，给定阈值$minSup$，若子图$g \in G$在图集合G中出现的频率（图集合的每个子图只计算1次）大于阈值$minSup$，则子图g为频繁子图。

gSpan算法流程如算法1所示。

算法 1: gSpan算法主框架

Input: 图集合G, 阈值$minSup$
Output: 频繁子图

```
Function gSpan(G, minSup):
    对G中所有的顶点和边按照出现的频率从大到小重新标号;
    计算G中的频繁边，记为FreqEdge;
    删除G中不在FreqEdge中的边;
    对FreqEdge中的边按照频率从大到小排序;
    for 边 e ∈ FreqEdge do
        SubMining(G,e,FreqEdge,minSup);
        从G中删除所有的e;
        FreqEdge = FreqEdge − e;
    end
end
```

算法 2: gSpan算法SubMining过程

Input: 图集合G, 阈值$minSup$, 频繁边集和$FreqEdge$, 当前DFSCODE$dfscode$

Output: 频繁子图

Function *SubMining(G,dfscode,FreqEdge,minSup)*:
  构建一条边（根据规则正行或反向）的DFSCODE df;
  $dfscode = dfscode + df$;
  if $dfscode$是最小$DFSCODE$ **then**
    计算$dfscode$的频率Sup;
    if $Sup \geq minSup$ **then**
      SubMining(G,$dfscode$,$FreqEdge$,$minSup$);
    end
    return;
  end
end

§ 2 程序设计

2.1 图-类：class Graph

2.1.1 定义代码

```
struct Vertex
{
    int id;
    int label;
    int seq;
    bool del;

    Vertex(int _id = 0, int _label = 0) : id(_id), label(_label), seq(-1),
        del(0) {}
    ~Vertex() {}
};

struct Edge
{
    int u;
    int v;
```

```
16      int label;
17      int next;
18      bool del;
19
20      Edge(int _u = 0, int _v = 0, int _label = 0, int _next = -1) : u(_u),
            v(_v), label(_label), next(_next), del(0) {}
21      ~Edge() {}
22
23      bool operator == (const Edge &o) const
24      {
25          return u == o.u&&v == o.v&&label == o.label;
26      }
27  };
28
29  class Graph
30  {
31  public:
32      Graph()
33      {
34          memset(head, -1, sizeof(head));
35          vn = 0;
36          en = 0;
37      }
38      ~Graph() {}
39
40      void init();
41      void addv(int id, int label);
42      void addse(int u, int v, int label);
43      void adde(int u, int v, int label);
44      void delse(int u, int v, int label);
45      void dele(int u, int v, int label);
46
47  public:
48      const static int maxv = 250;
49      const static int maxe = 510;
50
51  public:
52      int head[maxv];
53      int vn;
54      int en;
55      Vertex vtx[maxv]; // 0 to vn-1
56      Edge edge[maxe]; // 0 to en-1
57  };
```

2.1.2 设计

图的存储使用链式前向星来存储。链式前向星的效率高于使用$vector$写的邻接表。

链式前向星的标准设计是：

- $head$[]数组：大小为顶点数，存这个点的对应的第一条边在$edge$[]数组的下标
- $edge$[]数组：用数组存储边
- $Edge$边的结构：边的节点u, v，边的标号$label$，删除标记del，下一个访问的边的下标$next$
- 添加边：

```
void Graph::addv(int id, int label)
{
    vtx[id] = Vertex(id, label);
    vn++;
}
void Graph::addse(int u, int v, int label)
{
    edge[en] = Edge(u, v, label, head[u]);
    head[u] = en++;
}
```

- 访问一个定点的所有边

```
for (int i = head[u]; ~i; i = edge[i].next)
{
    Edge e = edge[i];
    // Solve this edge
}
```

- 删除边：令边的$del = 1$，由于边i和边i^1互为反向边，所以直接遍历一次即可。

```
void Graph::dele(int u, int v, int label)
{
```

```
        for (int i = head[u];~i;i = edge[i].next)
            if (edge[i].u == u&&edge[i].v == v&&edge[i].label == label)
            {
                edge[i].del = 1;
                edge[i ^ 1].del = 1;
                return;
            }
    }
```

图里面用数组存储节点的访问顺序和点的标号，用链式前向星存储图的结构。

2.2 输入和重排序-类：class InputFilter

2.2.1 定义代码

```
1  class InputFilter
2  {
3  public:
4      struct Node
5      {
6          int label;
7          int cnt;
8          Node(int _label = 0, int _cnt = 0) : label(_label), cnt(_cnt) {}
9          bool operator < (const Node &o) const // greater
10         {
11             return cnt > o.cnt;
12         }
13     };
14
15 public:
16     void init();
17     void addv(int id, int label);
18     void adde(int u, int v, int label);
19     void filterV();
20     void filterE();
21     void filter();
22
23 public:
24     const static int maxv = 250;
25     const static int maxe = 510;
26
27 public:
```

```
28      int cntv[maxv], cnte[maxe];
29      int mpv[maxv], mpe[maxe];
30      vector<Vertex> vecv;
31      vector<Edge> vece;
32      vector<int> listv, liste ;
33      vector<Node> filterv, filtere ;
34      vector<string> inputStr;
35  };
```

2.2.2 设计

数据设计

- $int\ cntv[maxv], cnte[maxe]$：点和边的出现次数计数
- $int\ mpv[maxv], mpe[maxe]$：重标号后点和边对应的标号
- $vector < Vertex > \ vecv$：存储所有的点
- $vector < Edge > \ vece$：存储所有的边
- $vector < int > \ listv, liste$：存储点和边的下标
- $vector < Node > \ filterv, filtere$：用于对点和边按照频度进行排序
- $vector < string > \ inputStr$：将文件输入存储下来，后面用$inputStr$将数据输入到$GSPAN$中

流程设计

- $void\ filterV()$：对点进行重标号，统计点的频度，然后对点的频度进行排序，将映射结果记录在$mpv[]$中
- $void\ filterE()$：对边进行重标号，统计边的频度，然后对边的频度进行排序，将映射结果记录在$mpe[]$中
- $void\ filter()$：对点和边进行重标号，直接调用filterV()和filterE()

2.3 DFS Code五元组节点-结构体：struct DFSCodeNode

2.3.1 定义代码

```
1  struct DFSCodeNode
2  {
3      int a, b;
4      int la, lab, lb;
5
6      DFSCodeNode(int _a = -1, int _b = -1, int _la = -1, int _lab = -1,
           int _lb = -1) : a(_a), b(_b), la(_la), lab(_lab), lb(_lb) {}
7      ~DFSCodeNode() {}
8
9      bool isForward() const
10     {
11         return a < b;
12     }
13     bool isBackward() const
14     {
15         return a > b;
16     }
17
18     bool operator < (const DFSCodeNode &o) const
19     {
20         if (this->isBackward() && o.isForward()) return 1;
21         else if (this->isBackward() && o.isBackward() && b < o.b)
               return 1;
22         else if (this->isBackward() && o.isBackward() && b == o.b&&
               lab < o.lab) return 1;
23         else if (this->isForward() && o.isForward() && a > o.a) return
               1;
24         else if (this->isForward() && o.isForward() && a == o.a&&la
               < o.la) return 1;
25         else if (this->isForward() && o.isForward() && a == o.a &&la
               == o.la&&lab < o.lab) return 1;
26         else if (this->isForward() && o.isForward() && a == o.a &&la
               == o.la&&lab == o.lab&&lb < o.lb) return 1;
27         return 0;
28     }
29     bool operator == (const DFSCodeNode &o) const
30     {
31         return a == o.a&&b == o.b&&la == o.la&&lab == o.lab&&lb
               == o.lb;
32     }
33 };
```

2.3.2 设计

数据设计 这里的五元组(a, b, la, lab, lb)为DFS编码五元组的(i, j, l_i, l_{ij}, l_j)

流程设计

- $bool\ isForward()$：是否为正向边，直接判断是否$a < b$
- $bool\ isBackward()$：是否为反向边，直接判断是否$b < a$
- 重载==号和<号，<号按照DFScode的小于号定义来写

2.4 DFS Code-类：class DFSCode

2.4.1 定义代码

```
1  class DFSCode
2  {
3  public:
4      DFSCode()
5      {
6          dfsCodeList.clear();
7          rightPath.clear();
8      }
9      bool operator < (const DFSCode &o) const
10     {
11         int minsize = min(dfsCodeList.size(), o.dfsCodeList.size());
12         for (int i = 0;i < minsize;i++)
13             if (dfsCodeList[i] < o.dfsCodeList[i]) return 1;
14         return dfsCodeList.size() < o.dfsCodeList.size();
15     }
16     bool operator == (const DFSCode &o) const
17     {
18         if (dfsCodeList.size() != o.dfsCodeList.size()) return 0;
19         for (int i = 0;i < (int)dfsCodeList.size();i++)
20             if (!(dfsCodeList[i] == o.dfsCodeList[i])) return 0;
21         return 1;
22     }
23
24 public:
25     void init();
26     void output(); // Output this dfscode
27     Graph Convert2Graph();
28     bool GenMinDFSCode(Graph &g, DFSCode &ret, int now); //
           Generate the min dfscode from now
29     DFSCode FindMinDFSCode(); // Find the min dfscode of this pattern
```

```
30      bool isMinDFSCode(); // Is this dfscode the min dfscode?
31
32  public:
33      vector<DFSCodeNode> dfsCodeList;
34      vector<pair<int, int> > rightPath;
35  };
```

2.4.2 设计

数据设计

- $dfsCodeList$：该DFSCODE，用vector存储五元组
- $rightPath$：该DFSCODE的最右路径，$rightPath[rightPath.size() - 1]$即为最右节点

流程设计

- $Convert2Graph$：将该DFSCODE转化为图
  - 输入：该DFSCODE
  - 输出：该DFSCODE对应的链式前向星表示的图
  - 流程：直接按照dfsCodeList的dfscode序列插入点和边
- $GenMinDFSCode$：用DFS生成最小DFSCODE
  - 输入：DFS过程传递参数：g（当前的图，顶点带有访问序列编号），ret（当前的DFSCODE），now（从当前点扩展）
  - 输出：最小DFSCODE
  - 流程：
    1. 选取当前节点
    2. 在图中从当前节点找它的所有可能的正向边，并排序选取最小的正向边加入DFSCODE
    3. 在图中找到通过加入正向边的新节点的所有反向边，排序，按照从小到大的顺序插入DFSCODE
    4. 若构建的DFSCODE大于之前找到的DFSCODE，return；否则继续1-3的流程。
    5. 重复1-4流程，直至生成可行的DFSCODE
- $FindMinDFSCode$：寻找最小DFSCODE，直接调用$GenMinDFSCode$

- $isMinDFSCode$：判断该DFSCODE是否为最小DFSCODE，判断$FindMinDFSCode$找到的最小DFSCODE是否和该DFSCODE相同

2.5 gSpan算法-类：class GSPAN

2.5.1 定义代码

```
1  class GSPAN
2  {
3  public:
4      struct FreqEdgeSortNode
5      {
6          Edge e;
7          int cnt;
8          FreqEdgeSortNode(Edge _e = Edge(), int _cnt = 0) :e(_e), cnt(_cnt)
             {}
9          bool operator < (const FreqEdgeSortNode &o) const // greater
10         {
11             return cnt > o.cnt;
12         }
13     };
14
15 public:
16     GSPAN() {}
17     void init();
18     void input(const InputFilter &_inputFilter, double _minSup); // Build
           relabeled graph
19     void output();
20     void GenSeedSet(); // Generate the seed edge set
21     void DeleteEdgeFlag(const Edge &e); // Label deleted edge
22     void DeleteEdge(const Edge &e); // Delete edge from graph
23     void DeleteUnFreqEdge(); // Delete unfreq edge
24     void RebuildGraph(int id); // Rebuild graph with id
25     bool JudgePatternInGraph(Graph &graph, const DFSCode &dfscode,
           int ith, int now); // DFS, ith = dfscode.dfsCodeList[ith], now =
           now vertex
26     bool isPatternInGraph(Graph graph, const DFSCode &dfscode); // Is
           this pattern in this graph?
27     void SolveFreqPattern(const DFSCode &dfscode); // Work when
           dfscode is freq pattern
28     bool isFreqPattern(const DFSCode &dfscode); // Is dfscode a freq
           pattern?
29     void BuildPattern(DFSCode &dfscode, int loc, int backloc, int maxseq);
           // DFS build pattern and test, loc = now extend location in
           rightpath, backloc = -1(forward) or backward location in rightpath,
           maxseq = max sequence id
30     void SubMining(const Edge &base); // Sub-Mining Procedure
```

```
31      void gSpan(); // Run gSpan
32
33      void debug(); // For debug
34
35  public:
36      const static int maxGraph = 10010; // Maximum graph number of
            graph set
37
38  public:
39      ofstream out; // Output to file
40      DFSCode tmpDFSCode; // Temp dfscode
41
42      double minSup; // minimum support
43      int minSupDeg; // minSup * cntGraph
44
45      Graph graph[maxGraph]; // 0 to cntGraph-1
46      int cntGraph; // Num of graphs in the graph set
47
48      map<Edge, int, EdgeCMP> freqEdgeCnt; // Count edge's frequency
49      set<Edge, EdgeCMP> freqEdgeVis; // Visit or not in a graph
50
51      vector<Edge> freqEdge; // Freq edge set
52      vector<Edge> unFreqEdge; // Unfreq edge set
53
54      vector<DFSCode> freqPattern; // Freq pattern, the answer
55  };
```

2.5.2 设计

- $ofstream\ out$：文件输出流
- $DFSCode\ tmpDFSCode$：临时变量，临时的DFSCode 变量
- $double\ minSup$：给定的$minSup$阈值$\in [0,1]$
- $int\ minSupDeg$：由于double变量存在精度误差，所以令$minSupDeg = (int)ceil(minSup \times cntGraph)$，将阈值转化为int值频度
- $Graph\ graph[maxGraph]$：图集合
- $int\ cntGraph$：图集合的大小
- $map < Edge, int, EdgeCMP > freqEdgeCnt$：临时变量，用于计算频繁边出现的频度

- $set < Edge, EdgeCMP > \ freqEdgeVis$：临时变量，用于记录此边是否访问过
- $vector < Edge > \ freqEdge$：频繁边集合
- $vector < Edge > \ unFreqEdge$：不频繁边集和
- $vector < DFSCode > \ freqPattern$：最终的频繁模式，即最终答案

流程设计

- $init$：初始化
- $input$：读入图集合数据
- $output$：输出最终的挖掘结果
- $GenSeedSet$：生成初始频繁边集合
  - 输入：无
  - 输出：初始频繁边集合
  - 流程：
    1. 使用map记录每种边的频度
    2. 频度大于阈值的边即为频繁边
    3. 按照从大到小将所有频繁边进行排序
- $DeleteEdgeFlag$：对某条边在图集合中标记删除符号
  - 输入：边
  - 输出：标记删除符号后的图集合
  - 流程：遍历整个图集合，对此边标记$del = 1$
- $DeleteEdge$：删除某条边，调用$DeleteEdgeFlag$，再调用$RebuildGraph$重构整个图集合
- $DeleteUnFreqEdge$：删除所有不频繁的边，对$unFreqEdge$里面的元素调用$DeleteEdgeFlag$后，再调用$RebuildGraph$重构整个图集合
- $RebuildGraph$：重构整个图集合
  - 输入：带删除标记的图集合
  - 输出：重构后的图集合

  - 流程：直接使用不带删除标记的点和边构建新的图集合

- $JudgePatternInGraph$：
  - 输入：dfscode，一个图
  - 输出：此dfscode是否在图中出现
  - 流程：
    1. 找到起始顶点（标号和dfscode的0一样）集合；
    2. 对于起始顶点，直接按照dfscode的顺序在图上进行dfs，如果能够完全通过整个dfscode，则出现，否则没有出现；
    3. 对于起始顶点的每个顶点执行2。
- $isPatternInGraph$：判断该模式是否在某图中出现，直接调用$JudgePatternInGraph$。

- $SolveFreqPattern$：处理该频繁模式，直接将它插入到结果集合中。
- $isFreqPattern$：判断是否为频繁模式
  - 输入：DFSCODE
  - 输出：是否为频繁模式
  - 流程：
    1. 对于图集合的每一个图，调用$isPatternInGraph$，计算频率sup
    2. 如果$sup \geq minSup$，返回$TRUE$，否则返回$FALSE$。
- $BuildPattern$：DFS构建一个模式并判断是否为频繁模式
  - 输入：当前的DFSCODE；loc是当前在rightpath中扩展节点的下标；backloc为-1表示进行扩展正向边，为大于等于0的数表示应加入的反向边终点在rightpath中的位置；maxseq表示当前最大的访问顺序标号。
  - 输出：判断频繁模式，以及下一个DFSCODE
  - 流程：
    1. $backloc == -1$?
       * $backloc == -1$，进行正向边扩展
         (a) 查找loc对应的节点可行的正向边扩展
         (b) 加入正向边，构造新的$dfscode$
         (c) 判断是否该$dfscode$频繁且是最小DFSCODE

· 是，则判断：如果$rightpath.size() < 3$，表明没有反向边可加入，调用$BuildPattern(dfscode, (int)dfscode.rightPath.size()-1, -1, maxseq+1)$继续加正向边；如果$rightpath.size() \geq 3$，表明有反向边可加入，调用$BuildPattern(dfscode, t-1, 0, maxseq)$，加入反向边。

· 否，则根据$rightpath$，向祖先节点查找可以加入正向边的点，并同步更新$rightpath$，然后调用$BuildPattern(dfscode, (int)dfscode.rightPath.size()-1, -1, maxseq)$从此rightpath上的祖先节点扩展正向边

* $backloc != -1$，进行反向边扩展

(a) 对于每一个与$rightpath[loc]$和$rightpath[backloc]$顶点标号相同的边，构造五元组，加入当前$dfscode$

(b) 判断是否该$dfscode$频繁且是最小DFSCODE

(c) 如果$loc-backloc < 2$，则调用$BuildPattern(dfscode, loc, -1, maxseq)$进行正向边扩展，否则调用$BuildPattern(dfscode, loc, backloc+1, maxseq)$继续扩展反向边。

- $SubMining$：对于一个频繁边，进行挖掘
  - 输入：频繁边
  - 输出：挖掘出的频繁模式
  - 流程：对于每一条边，以其为起点，调用$BuildPattern$，DFS寻找频繁模式
- $gSpan$：gSpan算法主流程
  - 输入：图集合G，阈值$minSup$
  - 输出：频繁子图集合
  - 流程：
    1. 调用$input$读取被$InputFilter$预处理好的图集合
    2. 调用$GenSeedSet$生成频繁边集合
    3. 调用$DeleteUnFreqEdge$删除不频繁边
    4. 对于$freqEdge$每一个元素，调用$SubMining$
    5. 调用$DeleteEdge$在图集合中删除该边
    6. 对于$freqEdge$每一个元素执行4-5步

§ 3 程序测试

3.1 测试环境

测试环境如表1所示：

表 1 实验环境

项目	详细信息
CPU	AMD Opteron 8380 (2.5GHz, 4 Cores) × 16
内存	64GB ECC DDR2
测试所用磁盘	2TB 7200RPM HDD (Read: 96.5MB/s)
操作系统	Ubuntu 12.04.2 LTS x64
C/C++编译器	GNU C++ 4.8

该代码已上传至BitBucket的私有仓库，课程结束后会开源。

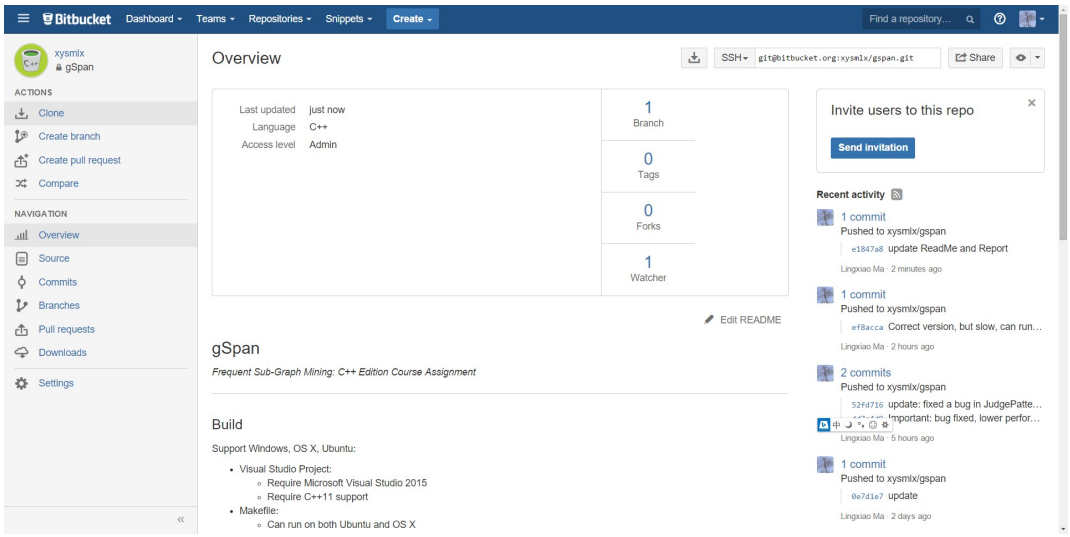


图 1 BitBucket私有仓库

该代码在Windows 10、OS X Yosemite、Ubuntu 12.04.2 LTS下分别用Visual Studio、Clang++、G++编译通过，可跨平台。

3.2 正确性测试

3.2.1 测试数据

测试数据集为网上的博客的一个展示[1]。

[1]数据来源：http://simpledatamining.blogspot.ca/2015/04/how-to-mine-frequent-patterns-in-graphs.html

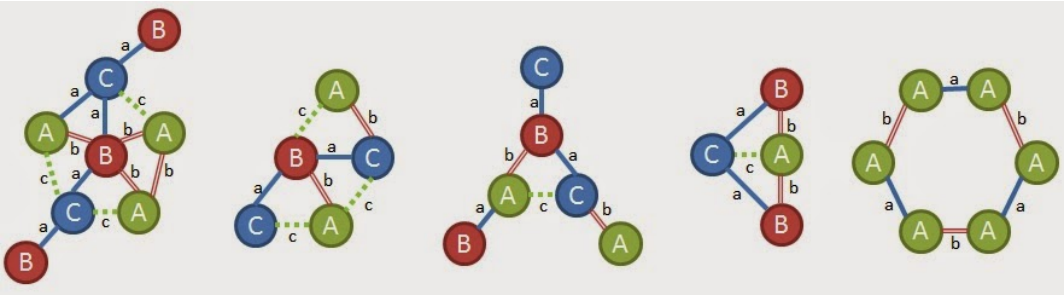


图 2 正确性测试的测试数据

```
1  t # 0
2  v 0 3
3  v 1 4
4  v 2 2
5  v 3 3
6  v 4 2
7  v 5 2
8  v 6 4
9  v 7 3
10 e 0 1 2
11 e 1 2 2
12 e 2 3 3
13 e 1 3 2
14 e 3 4 3
15 e 1 4 4
16 e 4 5 3
17 e 3 5 3
18 e 5 6 4
19 e 3 6 2
20 e 6 2 4
21 e 6 7 2
22 t # 1
23 v 0 2
24 v 1 3
25 v 2 4
26 v 3 2
27 v 4 4
28 e 0 1 4
29 e 1 2 2
30 e 0 2 3
31 e 2 3 4
32 e 1 3 3
33 e 3 4 4
34 e 1 4 2
```

```
t # 2
v 0 4
v 1 3
v 2 2
v 3 3
v 4 4
v 5 2
e 0 1 2
e 1 2 3
e 2 3 2
e 2 4 4
e 1 4 2
e 4 5 3
t # 3
v 0 3
v 1 2
v 2 3
v 3 4
e 0 1 3
e 1 2 3
e 2 3 2
e 3 0 2
e 3 1 4
t # 4
v 0 2
v 1 2
v 2 2
v 3 2
v 4 2
v 5 2
e 0 1 2
e 1 2 3
e 2 3 2
e 3 4 3
e 4 5 2
e 5 0 3
t # -1
```

3.2.2 输出结果

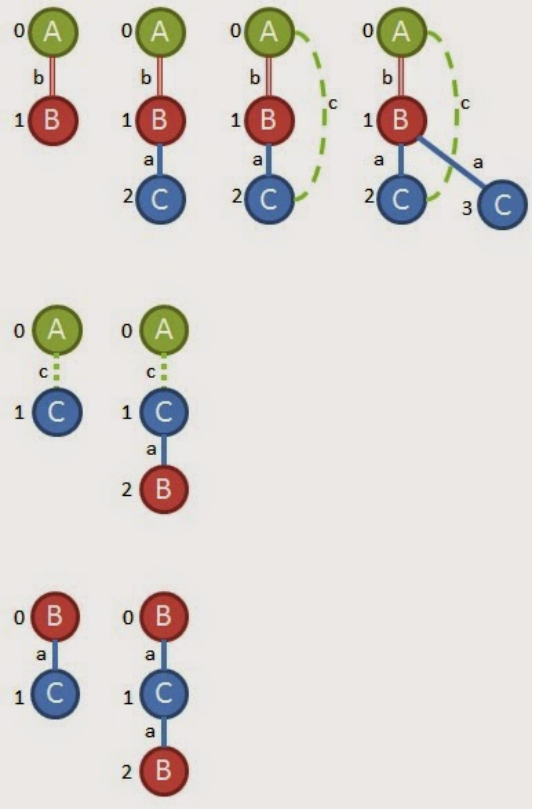


图 3 正确性测试的结果

```
1  Seed Set:
2  2 3 3
3  2 4 4
4  3 4 2
5  Pattern #1:
6  0 1 2 3 3
7  Pattern #2:
8  0 1 2 3 3
9  1 2 3 2 4
10 Pattern #3:
11 0 1 2 3 3
12 1 2 3 2 4
13 2 0 4 4 2
14 Pattern #4:
15 0 1 2 3 3
16 1 2 3 2 4
17 2 0 4 4 2
18 1 2 3 2 4
19 Pattern #5:
```

```
0 1 2 4 4
Pattern #6:
0 1 2 4 4
1 2 4 2 3
Pattern #7:
0 1 2 4 4
1 2 4 2 3
2 2 3 2 4
Pattern #8:
0 1 3 2 4
```

3.3 运行速度测试

3.3.1 测试数据

测试数据为graph.data：图集合中有10000个图，每个图最大点数不超过250，边数不超过250。

3.3.2 测试结果

使用Xifeng Yan, Jiawei Han的原作者的程序[2]进行运行速度测试的对比，*minSup*阈值选取和运行时间数据如表2所示。运行时间作图如图4所示。

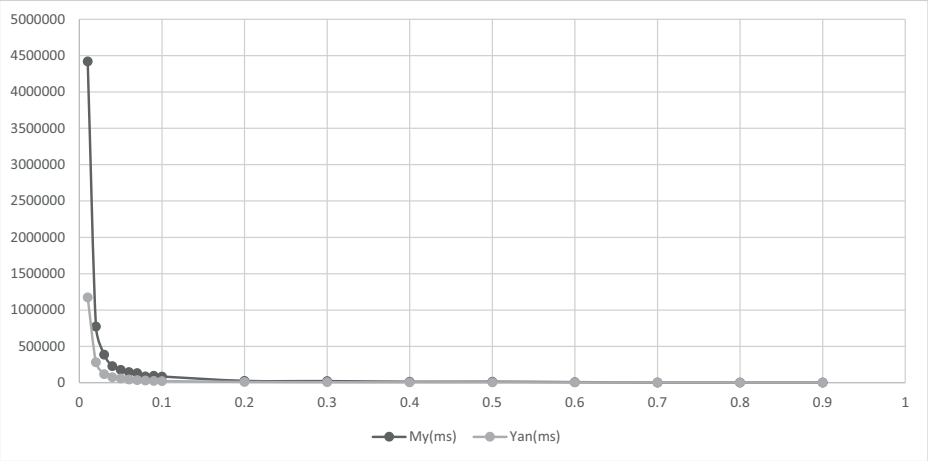


图 4 运行时间测试

[2] http://www.cs.ucsb.edu/~xyan/software/gSpan2009-02-20/gSpan6.tar.gz

表 2 运行时间测试

minSup	My(ms)	Yan(ms)
0.9	2050	1837
0.8	2790	2299
0.7	3520	4298
0.6	8510	4554
0.5	14920	5749
0.4	13690	6001
0.3	22620	7318
0.2	25940	11164
0.1	85820	23648
0.09	96650	26878
0.08	87780	31235
0.07	133020	36047
0.06	146220	45141
0.05	178760	58311
0.04	227410	75880
0.03	387420	119003
0.02	772316	282170
0.01	4421358	1175110

从表2和图4看出，本程序的性能不如原作者的程序的性能。还有一些优化可以加入，但是因为时间的原因，所以没有加入程序。

§ 4 总结

通过此次实验，完整的编写gSpan算法，对频繁子图挖掘和DFSCODE理论有了更深的理解。在通读Xifeng Yan的论文后，发现整个gSpan算法的框架其实是比较naive的：找到频繁边然后让边扩展再在图集合中匹配。然而在我看来，这篇论文的发光点以及核心是DFSCODE理论，正是DFSCODE使得gSpan的性能明显优于当时最好的频繁子图挖掘算法FSG。

此程序的性能不如原作者的程序，不过由于时间原因，还有一些想到的优化策略没有加入程序，期望加入程序后再次和原作者的程序比一比。

优化策略

- 链式前向星写的图的点数和边数开的较大，可以根据图的大小开；

- 将枚举构造频发模式改为在图集合中查找构造频繁模式；
- gSpan有很好的并行性，所以可以使用多线程：将每个任务放入一个队列，然后让每个线程从队列中提取任务执行；
- gSpan有很好的并行性，所以可以使用MapReduce等分布式框架[2]或者GPU计算[3]。

参考文献

[1] Yan, Xifeng, and Jiawei Han. "gspan: Graph-based substructure pattern mining." Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on. IEEE, 2002.

[2] Lin, Wenqing, Xiaokui Xiao, and Gabriel Ghinita. "Large-scale frequent subgraph mining in mapreduce." Data Engineering (ICDE), 2014 IEEE 30th International Conference on. IEEE, 2014.

[3] Kessl, Robert, et al. "Parallel Graph Mining with GPUs." The 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications. 2014.