Graph Pattern Matching

Professor Zhijia Zhao University of California, Riverside

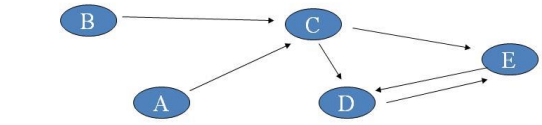
Tanaya Vadgave University of California, Riverside

***Abstract*— Graph Pattern Matching is a generalization of string matching and 2D pattern matching that offers a natural framework for the study of matching problems upon multidimensional structures. This report presents basic terminology, graph coloring, graph homomorphism, implementation of basic search algorithm in C++, parallelization of search algorithm and idea of speculative tree traversal in graph pattern matching.**

I. Introduction

Graph Pattern Matching is a generalization of string matching and 2D pattern matching that offers a natural framework for the study of matching problems upon multidimensional structures. Before we move ahead with Graph pattern matching algorithm, let us see few related terminology.

A. Graph Representation Using Adjacency Matrix



adj

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

• Represents the fact that an edge exists between the two Nodes.

Figure 3 : Adjacency Matrix

B. Graph Isomorphism

Graph G to graph H is a bijective mapping α: v(G)->v(H) such that, xyϵE(G)<-> α(x) α(y) ϵE(H).

1. Edges to edge from graph G to graph H are mapped by function α.
2. Non-edges to non-edges are mapped by function α.

That is, isomorphism preserves adjacency and non-adjacency.

C. Graph Homomorphism

Graph homomorphism is mapping between 'any 'two graphs (irrespective of pattern/target) that respects their structure.
From graph G to graph H, α: v(G)->v(H) such that, xyϵE(G)<-> α(x) α(y) ϵE(H).
In this method, mapping is not necessarily bijective. Meaning, adjacency is preserved but non-adjacency is not preserved.

1. Edges to edge from graph G to graph H are mapped by function α.
2. Non-edges are mapped to single vertex/edge/non-edge by function α.

Graph homomorphism does not care where the non-edges are getting mapped to simplifying computation. We will see the example through next concept- Graph Coloring.

D. Graph Coloring ,homomorphism and pattern matching

Graph coloring is special case of graph labelling. Consider example of vertex coloring shown in figure 4 where *no two adjacent vertices share same color*.
Consider following figure. Let G be the target graph and H be the pattern graph.

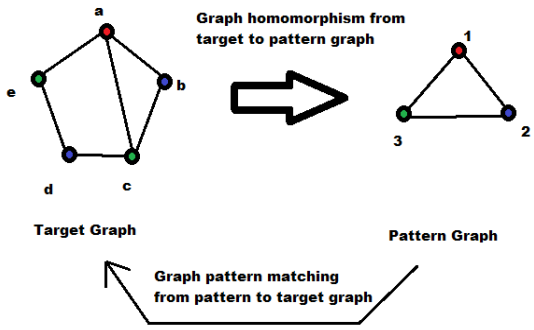


Figure 4 : Graph homomorphism and pattern matching

Now consider vertices in graph G and corresponding mapped vertices in H.

The relation α: v(G)->v(H) is given by:

Mapping of vertices in target graph to corresponding vertices in pattern graph=>

A	1 Red
B	2 Blue
C	3 Green
D	2 Blue
E	3 Green

Mapping of edge/non-edges in target graph to corresponding edges/vertices in pattern graph=>

Relation	Significance	Mapping into
a-b	Edge	1-2 Edge
b-c	Edge	2-3 Edge
c-d	Edge	3-2 Edge
d-e	Edge	2-3 Edge
e-a	Edge	3-1 Edge
a-c	Edge	1-3 Edge
e-b	Non-edge	3-2 Edge
b-d	Non-edge	2-2 vertex

Above tables explain properties of graph homomorphism we have seen in terminology 5.

1. Edges to edge from graph G to graph H are mapped by function α.
2. Non-edges are mapped to single vertex/edge/non-edge by function α.

E. Condition for Graph Homomorphism-

A relation $\Phi \subseteq V * V'$ is a homomorphism from a graph G = (V, B) to a graph G′= (V′, B′) if

$\Phi^T\Phi \subseteq I$, $I \subseteq \Phi\Phi^T$, $B \subseteq \Phi B'\Phi^T$;

and it is also written Φ: G→G′. (G is target graph and G' is pattern graph)

Example: The relation
Φ= {(a, x), (b, y), (c, x), (d, z), (e, z)}
is a homomorphism from the graph G = (V, B) with V = {a, b, c, d, e} and B = {(a, b), (b, b), (c, e), (d, a), (d, b), (e, c)} to the graph G′ = (V′, B′) with V′ ={x, y, z} and B′ = {(x; y), (x, z), (y, y), (z, x), (z, y)}:

$$B = \begin{array}{c|ccccc} & a & b & c & d & e \\ a & 0 & 1 & 0 & 0 & 0 \\ b & 0 & 1 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 & 1 \\ d & 1 & 1 & 0 & 0 & 0 \\ e & 0 & 0 & 1 & 0 & 0 \end{array} \quad \Phi = \begin{array}{c|ccc} & x & y & z \\ a & 1 & 0 & 0 \\ b & 0 & 1 & 0 \\ c & 1 & 0 & 0 \\ d & 0 & 0 & 1 \\ e & 0 & 0 & 1 \end{array} \quad B' = \begin{array}{c|ccc} & x & y & z \\ x & 0 & 1 & 1 \\ y & 0 & 1 & 0 \\ z & 1 & 1 & 0 \end{array}$$

F. Graph isomorphism vs Graph homomorphism-

In Subgraph isomorphism problem, more computations are required to map all edges and all non-edges to corresponding edges and non-edges. Hence, subgraph homomorphism problem is implemented in this project. If pattern graph has k connected components, then pattern matching along connected components requires worst case $\mathbf{O(m^{\frac{n}{k}})}$ complexity which is better than $O(m^n)$ of naïve algorithm. [Note: m=number of target nodes, n=number of pattern nodes.]

II. Basic Search Algorithm

A. Idea

Graph pattern matching is just like tree search. Let G = (V,B) be the pattern graph with n nodes. And G' = (V',B') be target graph with m nodes.
A graph homomorphism Φ:G->G' is given my n*m Boolean matrix satisfying condition for graph homomorphism discussed in terminology 6. Such a matrix will have exactly one true in each row, although it may have more than on true in each column.
[Reason: 2+ pattern nodes can be mapped into one target node. But one pattern node can not be mapped into 2+ target nodes.]
Any traversal of such tree could be used to enumerate all leaves and test them for homomorphism. An iterative search algorithm is given next.

B. Algorithm

1. Homomorphism function:

This is the function which takes pattern and target node as input and applies search algorithm to find homomorphism and pattern matching. P=Pattern matrix, T=Target matrix

```
function homomorphism(P, T)
return search(setup(P, T), P, T)
```

2. 'setup' function

This function sets homomorphism matrix H with all entries 1 initially.

```
function setup(P, T)
for i = 1 to n do
for j = 1 to m do
H[i, j]← true
end for
end for
return H
```

3. 'search' function

Search function traverses the tree in following way.
It first goes on updating homomorphism matrix H by row wise. When it reaches to leaf, it checks for condition for homomorphism. If satisfied, it returns the pattern graph.

Else it goes on updating homomorphism matrix H column wise.
This is shown in diagram below for n = 2 and m = 3. Tree traversal is shown by red arrows.

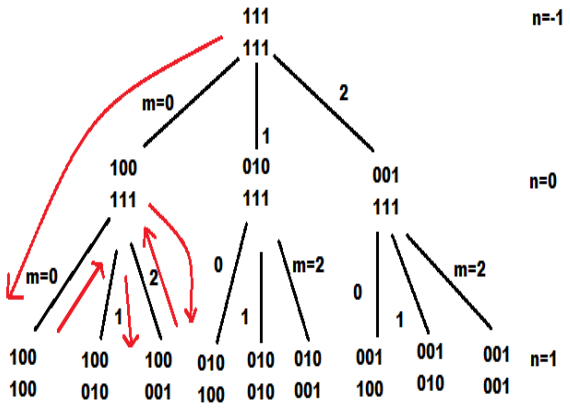


Figure 5: Tree Traversal

```
function search(H, P, T)
  if H is the one matrix then
  i← 0                    { build first leaf }
  else
  i← n                    { H is already a leaf, build next leaf}
  end if
  repeat

  if i < n then
  i← i + 1
  for j = 1 to m do
    H[i, j]← j = 1    { map node i to node 1}
  end for
  else
  while i ≥ 1 and H[i, m] do
  for j = 1 to m do
    H[i, j]← true  { restore i-th row }
  end for
  i← i - 1
  end while
  if i = 0 then
    H← null matrix
  else
  j← 1
  while not H[i, j] do
  j← j + 1
  end while
  H[i, j] ←false    { unmap node i to node j}
  H[i,j + 1]← true {map node i to node j + 1}
  end if
  end if
  until i = n and test(H, P, T) or i = 0
  return H
```

4. 'test' function

It implements condition for graph homomorphism discussed in terminology 6. '*I*' denotes identity Boolean matrix.

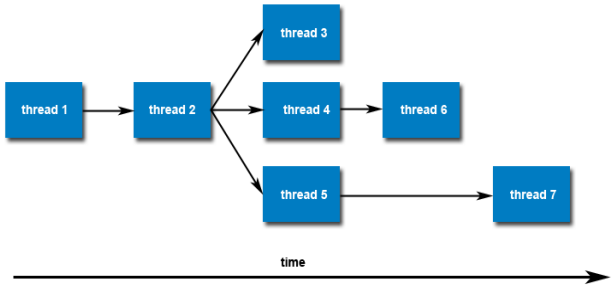
function test(H, P, T)
if $H^T \cdot H$ *is included in I then*
if I is included in $H \cdot H^T$ *then*
return P is included in $H \cdot T \cdot H^T$
end
end
return false

III. Implementation

Program is parallelized using pthreads. Pthread is a "procedure" that runs independently from its main program may best describe a thread. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library. [Later discussed in detail]

A. Changes made in code from milestone 2:

Library included: #include<pthread.h>
Library routine used: 'pthread_create' creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

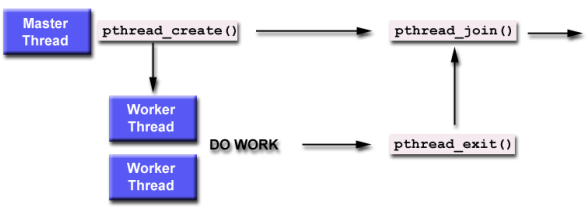


pthread_create function and its arguments in the code:

```
pthread_create(&threads[i],NULL,matrix_multiply, tparam)
```

Thread Joining:

"Joining" is one way to accomplish synchronization between threads. For example:



The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates. The function in the code is as follows:

```
pthread_join (threads[i], NULL);
```

Please refer following function to see the implementation of multi-threading:

```
1. float multiply_with_threads(long int **A,long int **B, int P, int Q)
2. void* matrix_multiply(void* params)
```

IV. Thread model in Graph Pattern Matching:

Observing previous milestone aim, we can say that multithreading has to be implemented though '**matrix-multiplication**'.

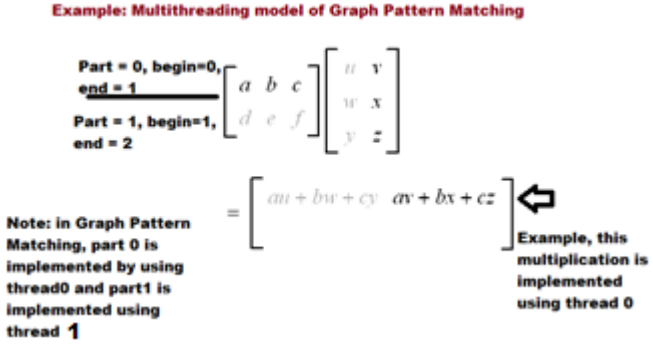
Initially, two threads are initialized. Threads count can be changed as variable is globally initialized. Using few computations, matrix is divided in two parts.

Statements representing this division in the matrix are:
int begin = (p*pattern_nodes)/num_threads;
int end = ((p+1)*pattern_nodes)/num_threads;

where, num_threads = number of threads
p = variable defined for matrix division
pattern_nodes = number of pattern graph nodes
'begin' and 'end' represent the outer limit of 2-D matrix multiplication.

For example in code, matrices 'H'(which is homomorphism matrix) and 'targetMatrix' are multiplied using a thread and result is updated in matrix 'mul' each time.
For(i=begin; i<end; i++)
For(j=0; j<target_nodes; j++)
For(k=0; k<target_nodes; k++)
Mul[i][j] += mul[i][j] + H[i][k]*targetMatrix[k][j];

Diagram:
patternMatrix = H*targetMatrix*Htranspose. Here, H is homomorphism matrix. Htranspose is transpose of H matrix
A simple 2-D matrix multiplication using thread is as follows:



V. Code compilation and Results

Following code compilation command has been used to compile the code consisting of PThreads.

Linux command: g++ -std=c++0x Graphs.cpp –lpthread

The code takes 'example.txt' file as an input. 'example.txt' consists of 4 sample examples which run one-by-one as the code execution starts. Make sure to add more examples (in any) in the same format as previous examples, else it will give error. Also make sure to keep the code and example.txt file in one folder for compilation purpose. Currently, example.txt file is in code folder.

'example.txt' has number of examples, number of target nodes, target graph edges, number of pattern nodes, pattern graph edges.

Final results:

```
EXAMPLE1
Enter number of target nodes:3
Enter Edge:1 1
Enter Edge:2 2
Enter Edge:-1 -1
Target Matrix is:
100
010
000
Enter number of pattern nodes:2
Enter Edge:1 1
Enter Edge:2 2
Enter Edge:-1 -1
Pattern Matrix is:
10
01
Pattern matrix found!!
Hence, The Homomorphism Matrix is:
100
010
```

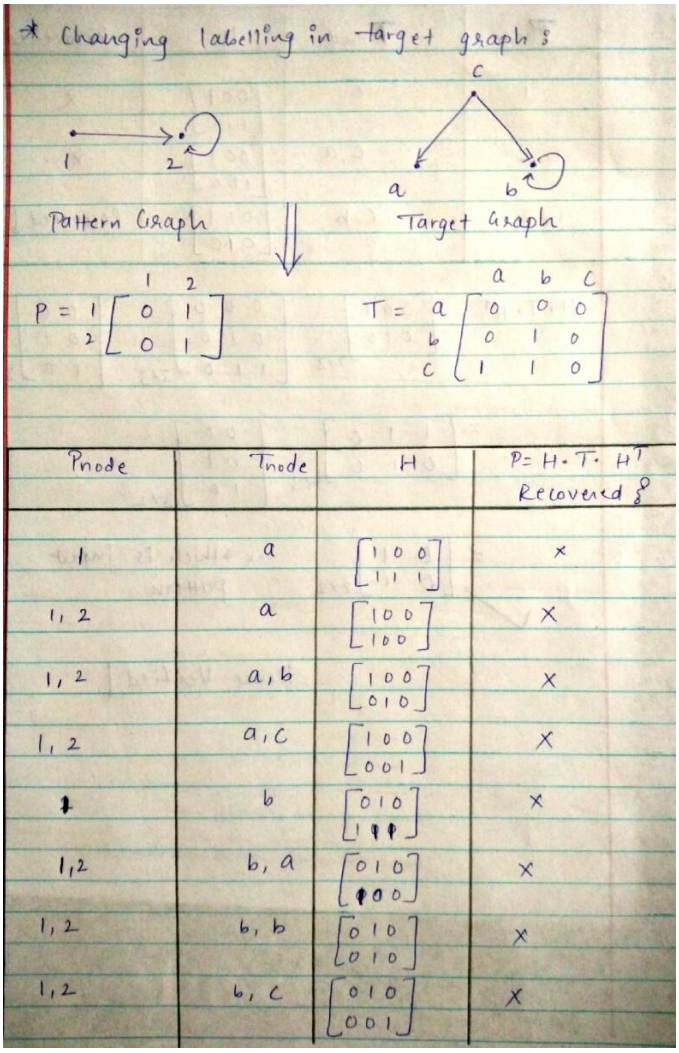
```
EXAMPLE2
Enter number of target nodes:3
Enter Edge:1 2
Enter Edge:2 3
Enter Edge:3 3
Enter Edge:-1 -1
Target Matrix is:
010
001
001
Enter number of pattern nodes:2
Enter Edge:1 2
Enter Edge:2 2
Enter Edge:-1 -1
```

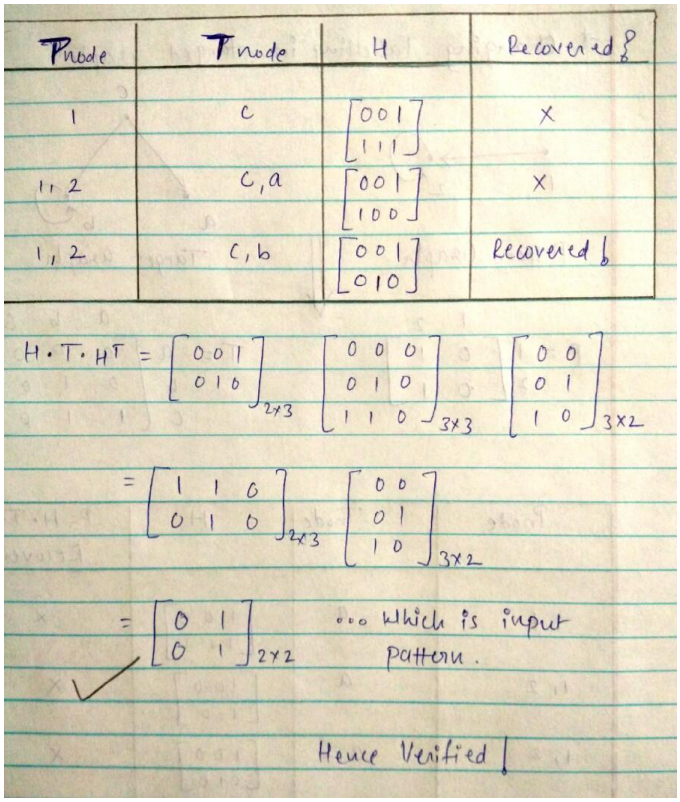
Pattern Matrix is:
01
01
Pattern matrix found!!
Hence, The Homomorphism Matrix is:
010
001

EXAMPLE3
Enter number of target nodes:3
Enter Edge:2 2
Enter Edge:3 1
Enter Edge:3 2
Enter Edge:-1 -1
Target Matrix is:
000
010
110
Enter number of pattern nodes:2
Enter Edge:1 2
Enter Edge:2 2
Enter Edge:-1 -1
Pattern Matrix is:
01
01
Pattern matrix found!!
Hence, The Homomorphism Matrix is:
001
010

EXAMPLE4
Enter number of target nodes:3
Enter Edge:1 2
Enter Edge:2 2
Enter Edge:1 3
Enter Edge:-1 -1
Target Matrix is:
011
010
000
Enter number of pattern nodes:2
Enter Edge:1 1
Enter Edge:1 2
Enter Edge:-1 -1
Pattern Matrix is:
11
00
Pattern Not Found!!

VI. On Paper Example run of example3





VII. Conclusion

The basic graph pattern matching has been successfully implemented using graph homomorphism. And it is successfully parallelized using pthreads.

References

[1] Gabriel Valiente "A Algorithm for Graph Pattern Matching"
[2] Wenfei Fan "Graph Pattern Matching: From Intractable to Polynomial Time"