XUN SUN

Computer Science Department
University of British Columbia

Finding Good Triangulation

-Implementation of the algorithm of "Finding Embeddings In A k-Tree"[1]

CPSC532C PROJECT REPORT

Dec. 17, 2004

Submitted to: Prof. Kevin Murphy

(Code accessible at: http://www.sunxun.com/ubc/532/ktree.zip)

1. General Description

1.1 Purpose

This project will implement the optimal triangulation algorithm mentioned in the paper "Complexity of Finding Embeddings In A *k*-tree", by Arnborg (1987)[1]. As shown in the paper that if we could find the *k*, then we can guarantee that the triangulated graph respect to the embedding *k*-tree will only include cliques with size no more than *k+1*.

It is shown in this paper that finding the embedding partial *k-tree* is an NP-Complete problem for a fixed size of *k*. The algorithm that is going to be implemented in this project, using dynamic program approach, has the complexity of $O(n^{k+2})$.

We have discussed the triangulation problem for an undirected graph in class. We have also implemented the triangulation using an arbitrary order in our assignment and we got to know that different elimination order could result in significant difference in the triangulated graph. The implementation of this algorithm will be handy when solving many graphic model related problems.

1.2 Assumptions

It is assumed that the input will be a 0/1 matrix representation of an undirected connected graph, and an integer *k* ($k>0$).

Because the algorithm to be implemented in this project has $O(n^{k+2})$ complexity, we assume that the *n* and *k* would be relatively small, otherwise, the running time is still very high. For example, if the input is a 10-by-10 grid matrix, which means $n=100$, and $k=10$, then the complexity in this case would be 10^{22}. It is obvious that the algorithm will perform poorly. The project will test the value of *n* and *k* that can be run in acceptable time.

In the paper, it doesn't mention how to generate the elimination order according to the partial *k*-tree embedding found by the algorithm. It is assumed here that the project could only return the cliques that are generated from each component of *k*-tree embedding, which guarantees that the maximum clique size will not be more than *k+1*, or it could run an algorithm that triangulate the graph with respect of these small *k*-tree embeddings, from smallest ones to bigger ones, and finally triangulate the entire graph. Because the purpose of finding the partial *k*-tree embedding is to limit the maximum clique size, it may not output a triangulation that has least fill-ins.

1.3 Method

The algorithm to finding if the given graph is a partial embedding *k*-tree uses dynamic programming method. Firstly, it finds all the *k*-vertex separators of the graph. Each such separator separates the graph to several connected components.

The mission is to test if each component is a partial *k*-tree. The base case is that a sub-graph with *k+1* vertices is a partial *k*-tree. In the dynamic programming process, the algorithm will fill out a table which is built to store the status of each connected component in the order of vertex size. If some component could be built by the union of the smaller components that have been proven to be partial *k*-trees, then this component could be partial *k*-tree and would be added to the partial *k*-tree list. The termination condition is that if for some separator, all its separated components are partial *k*-trees, and then the entire graph is a partial *k*-tree.

It is obvious that each smallest component could be a potential clique if we triangulate the graph. Therefore, in the process of dynamic programming, the clique composition is stored in another structure, so that when the graph is proven to be a partial *k*-tree, it can be traced back how it could be decomposed by the small cliques.

2. Achievements

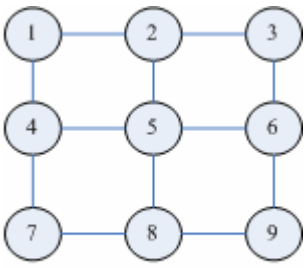
The implementation of the project strictly follows the algorithm description in the above mentioned paper. In addition, as mentioned in the proposal, the project will also output a structure which includes some information about the *k*-tree, the separators, the cliques, and the triangulated version of the graph with respect of each clique (atomic partial *k*-trees).

The project also tested the performance of the algorithm with the increase of the graph size and *k* value.

3. Project Documentation

3.1 User Interface

The project is implemented using Metalab 6.5. The process of finding *k*-tree embedding is built as a function. It can be called by any other programs. The input of the function is an undirected graph represented by a 0/1 matrix. For example:



Graph 3.1 Example of graph presentation

Can be represented as

```
0 1 0 1 0 0 0 0 0
1 0 1 0 1 0 0 0 0
0 1 0 0 0 1 0 0 0
1 0 0 0 1 0 1 0 0
0 1 0 1 0 1 0 1 0
0 0 1 0 1 0 0 0 1
0 0 0 1 0 0 0 1 0
0 0 0 0 1 0 1 0 1
0 0 0 0 0 1 0 1 0
```

The input value *k* will be a positive integer to be tested if the given graph can be a partial *k*-tree. The purpose is to find the minimum *k* such that the given graph could have a corresponding embedding *k*-tree.

3.2 Implementation

There are several main sub models along with the process of dynamic programming. Each main step is built as a sub-function, so that it could be easily tested and debugged. The main function is called *[Answer, RG] = pktree(G, k)*. The returned value includes: *Answer*, which is 0 or 1 indicating if the graph is a partial *k*-tree and a structure RG, which includes: *RG{1}* is the base separator of the graph, which can also be treated as the root graph of the resulting cliques; *RG{2}* is the entire set of cliques; *RG{3}* is the triangulated version of the original graph with respect to the partial embedding *k*-trees. It progressively calls the following main functions:

3.2.1 *Separators*

Prototype: [S,C]= separators(GT, K)

This function does the basic pre-procession to the input graph. It accepts the original graph and the *k* value. Firstly, all the combinations of *k*-vertex are built, which are potential "separator" that we need to test. Then, each potential separator is tested to see if deleting the separator the graph can be separated to several connected components. If it is such a separator, this function will also store the component graphs induced by this separator. Therefore, after running this function, proper separators and their corresponding components will be returned. The separators will become the base graph to build *k*-trees and the components are to be tested to see if it is also a partial *k*-tree or combination of partial *k*-trees.

This function also called three other smaller functions: *elimG(G, Elim)*, which will eliminate the vertices from original graph; *connected_g(G)*, which test if the graph is still connected after removing some vertices; *partition(G)*, which is used to generate the components after removing the base graph separator. Some other minors functions used include "*reachable(G,s)*", which uses breadth-first-search to find all the nodes that connect to vertex *s* , so that it could be judged if the given sub-graph is connected by the length of all reachable vertices.

3.2.2 *gsort*

Prototype: i_od= gsort(VCij)

This function sorts all the components by the order of their size. The input is the structure that includes all the components, which are represented by only the vertices numbers, not the graph matrix. It is because that we are only interested in the size here. It is easy to sort all the size numbers and the index of the separators. The numbers of each component are returned as a *N x 2* matrix, where *N* is the total amount of such components. Therefore, the returned matrix here is only the indexes of each C_i^j as defined in the algorithm.

3.2.3 select_sept

Prototype: select_sept(prot_Cm, S)

This function is used to recursively separate each component to smaller part and find separators in further components, until it reaches the components that only have *k+1* vertices. It firstly finds all the *k* vertices combinations in each component

and then selects those ones that are separators. Because we have the separators' set collection, we pass all the sets as a parameter to this function. In this way, the separators could be quickly found.

3.2.4 ktree_triangulate

Prototype: ktree_triangulate(UG, TAG)

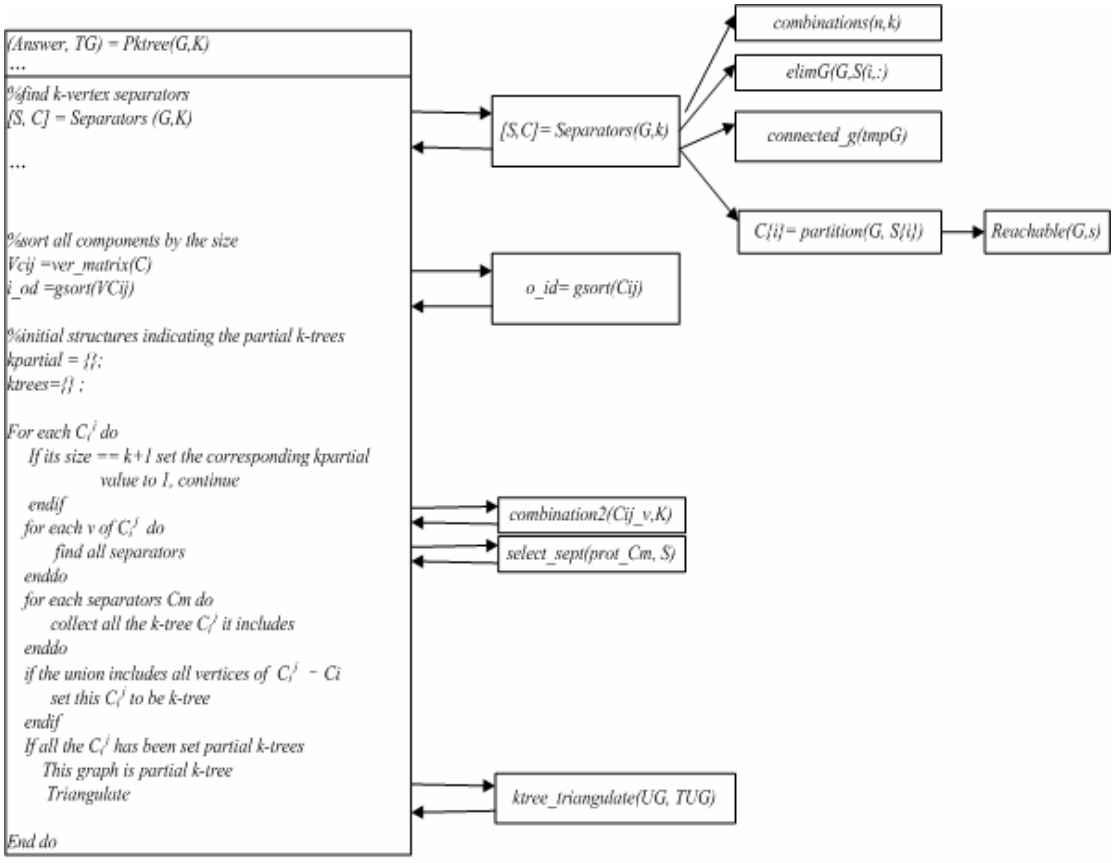
This function is used after the graph has been tested if it is a partial *k*-tree. It will do the triangulate part according to the partial *k*-trees. This function could only be called when the entire graph has been decomposed to small parts, and each part belongs to some *k*-tree embedding, which is also a potential clique. When the function is called, the entire function would have collected enough data structures, including the separators for each partial *k*-tree, the cliques, and the order that they are composed. Note that when the program does the dynamic programming procedure, it uses bottom-up approach to build those cliques, so that all the partial *k*-trees could be built by the combination of the cliques with at most *k+1* vertices.

3.2.5 Main Data Structures

There are several data structures used in this program, including:

- *S{}*: as defined in the algorithm, it includes all the *k*-vertex separators of the graph *G*. Each separator is represented as a cell of array.
- *C{}*: as C_i^j defined in the algorithm, it includes the matrix of each component. Note that it includes a series of cells, each of which includes cells of different components. The index of the first cell is in the same order with the index of *S*.
- *Kpartial{}:* It has the same structure with *C*. It is the indicator to each C_i^j , showing if it has been tested to be true (*k*-tree embeddable). It is initially set to be false to all the cells.
- *ktrees{}:* This structure is used to trace the clique composition of each component. When a component is verified to be a partial *k*-tree, the program will collect all the corresponding *k*-tree compositions of the component, which is either all the vertices when the size is *k+1,* or the *ktrees{i}* cells and the separators *S{i}* included.

The program function calling tree is shown as *graph 3.2*



Graph 3.2 Function Call Tree

3.3 Testing

The project is mainly in a format of function and it involves several sub-functions, therefore, the testing to each sub-function is performed to make sure that that the programs are all correct. Because the main algorithm is based on the above mentioned paper strictly, we can verify the correctness of the entire program by the output answer and the related *k*-tree information.

Several datasets are used for the testing of the program, including different size of graph and connectivity, such as *3x3* grids, *2x3* grids, graph used in assignment, and some graph with more vertices and different *k* values are also used to test the performance of the program.

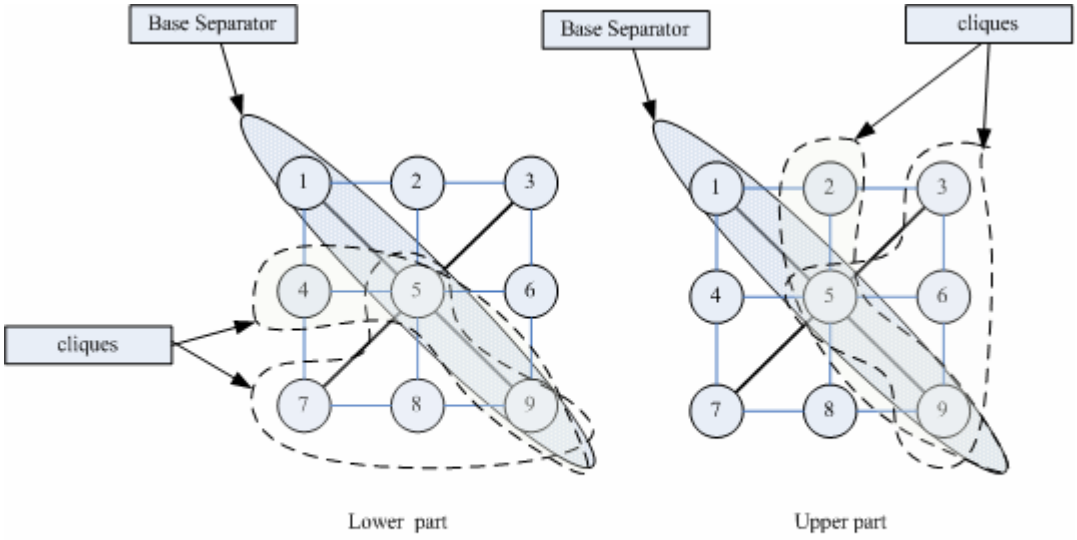
Here we only list the testing result made to *3x3* grid graph as an example to show how the testing is performed. Therefore, the input for the program is the matrix mentioned in graph 3.1 and *k* value of 3. *Note that more tests have been performed to make sure that the functions are correct. Only partial testing results are listed here for example and demonstration purpose only.*

3.3.1 Function Testing

Table 3.1 Function Testing Result
(with input of *3x3* grid G and *k=3)*

Function Name	Input	Output	Result
combinations(n,k)	*(length(G),3)*	*S, 84 x 3* elements, listing the 84 combinations of 3 numbers from 1 to *length(G)*	Pass
elimG(G,S)	*(G, S(8,:))*	*tempG*, a *6x6* matrix, resulted from G by delete the 1,3,4 lines and rows	Pass
connected_g(G)	*(tempG)*	*Yes*, when *1,3,4* are deleted *No*, when *1,2,6* are deleted	Pass
reachable(G,s)	*(tempG, 1)*	*No.*	Pass
partition(G, S)	*(G, S{4})*	C_i^j includes two *9x9* matrix, each represents part of the graph separated by *1,2, 6*	Pass
separators(G,k)	*(G,3)*	*[S, C] S* includes *36* cells; each includes a 3-vertex partition. C includes *36* cells also, each of which includes one or two cells of 9x9 matrix representing each separated graph.(deleted nodes have 0 to all other nodes)	Pass
ver_matrix(C)	*(C)*	*VCij* includes 36 cells, each has 2 elements and each element is a set of vertices (numbers)	Pass
combination2(V,k)	*([1 2 4 5],3)*	Protential_Cm includes *4* cells, each contains *3* numbers from input set	Pass
select_sept(Cm, S)	*(prot_Cm,S)*	*10,15* which are the indices of S, showing *S{10}* is a separator among the set	Pass
ktree_triangulate(UG, TUG)	*(G, TUG)*	*Triangulate*, includes the triangulated graph with respect to the partial *k-trees.*	Pass
Pktree(UG,k)	*(G,3)*	*1*, indicating that the graph is a partial *3-tree*. *TUG*, includes 3 parts, *{1}*is the base separator *1,5,9 ; {2}* includes all the cliques (partial *k*-tree components); *{3}* includes the triangulated graph	Pass

The resulting graph is shown as below *graph 3.3 :*



Graph 3.3 Triangulated Graph with Partial *K*-tree separators and resulting cliques:
{1,5,9},{2,5,9},{3,5,6,9}
{4,5,9},{5,7,8,9}

3.3.2 Performance Testing

To test the performance of the algorithm, Matlab functions *clock* and *etime* are used to record the start time, end time and their difference.

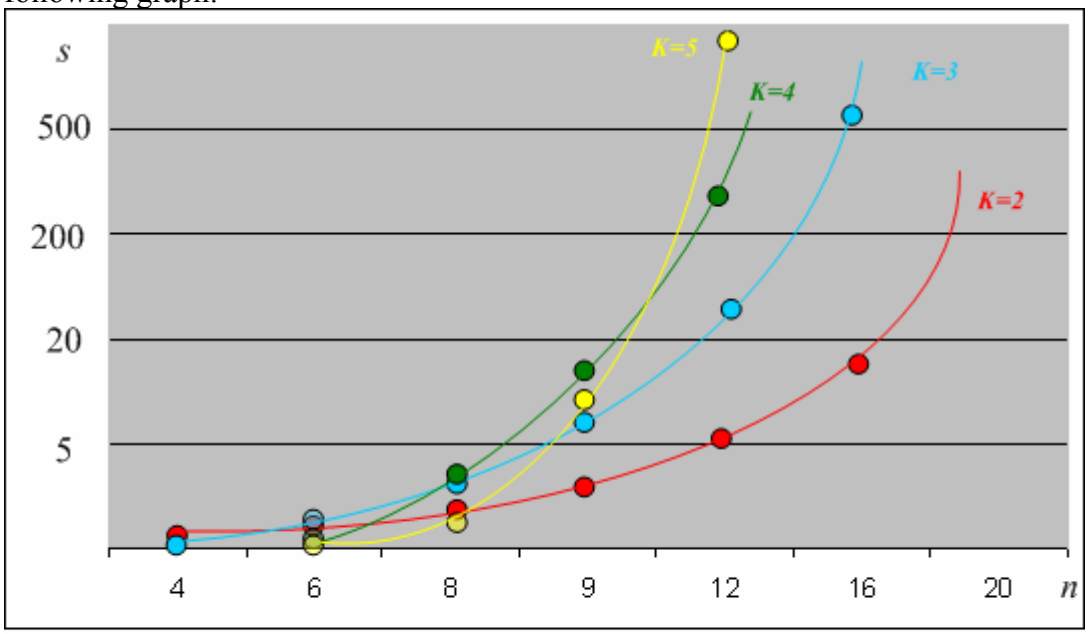
In a Pentium IV 2.0G CPU computer, the real running time comparison is as following:

Table 3.2 Program Running Time Testing Result

Number of Vertices	*K*	Partial *k*-tree?	Running Time (in second)	Number of Vertices	*K*	Partial *k*-tree?	Running Time (in second)
4	1	N	0.141	9	2	N	2.625
	2	Y	0.406		3	Y	7.891
	3	Y	0.313		4	Y	10.391
6	2	Y	0.984		5	Y	9.015
	3	Y	0.937	12	2	N	5.3600
	4	Y	0.343		3	N	180.3910
	5	N	0.062		4	Y	261.4370
8	2	Y	1.968		5	Y	711.1880
	3	Y	3.279		6	Y	330.5940
	4	Y	3.828	16	2	N	13.9850
	5	Y	2.562		3	N	581.8910
					4	Y	1502.032
					5	?	incomplete

The testing data above shows that the algorithm performs well while the size of the graph and *k* value is small. However, with the increasing of the size and *k*, the program performs very poorly. For a fixed *N* value, the running time increases with the size of *k*, until to [*n/2*]. This is because that in our program, the most expensive

part is to find the combinations of all the *k*-vertex separators. While calculating the combinations, we know that $C_n^k = C_n^{n-k}$, therefore, when *k=n/2*, the program would generate most number of different combinations. It execute the most time at this moment. In the testing process, when *n=16* and *k=5*, the program did not terminate in 3 hours. The relation of *n, k* and the running time is roughly shown in the following graph:



Graph 3.4 Program Running Time (Scaled)

X- number of vertices n

Y- running time (in second)

We can see that it roughly follows a n^{k+2} trend.

4. Conclusion

The algorithm in Arnborg[1] 's paper, which is successfully implemented in this project, is a way to find good triangulation such that the resulted graph could be guaranteed to have no cliques with size more than *k+1,* if the graph is verified to be a partial *k*-tree. The algorithm uses dynamic programming method, which improves the running time and the performance of the algorithm. However, as proved in the paper that the problem is a still *NP-complete* problem, and it costs n^{k+2} running time, therefore, it could almost be treated as exponential when the size of *n* and *k* increase to some extent.

There are many other later papers that use other algorithm to solve related problems, such as the algorithm in the paper *A Practical Relaxation of Constant-Factor Treewidth Approximation Algorithms* [2] by Hopins and Darwiche (2002) could guarantee that the triangulation of G with width at most *4k+1*; the paper *Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs* by Hans L. Bodlaender and Ton Kloks (1993) [5] described an O(*n*log *n)* algorithm to determine if tree-width ≤ *k*. The algorithm in this project is a good choice for small size graph. However, if we need process larger size graphs, we'd better select one of above mentioned algorithms.

5. Reference

[1] Arnborg, S., Corneil, D.G., and Proskurowski, A., *Complexity of Finding Embedding in a k- tree*, Journal of SIAM, Algebraic Discrete Methods, 8(2):177-184 (1987).

[2] Mark Hopkins and Adnan Darwiche, *A Practical Relaxation of Constant-Factor Treewidth Approximation Algorithms,* *http://reasoning.cs.ucla.edu/fetch.php?id=31&type=pdf*

[3] Mark A. Paskin and Gregory D. Lawrence (2003). *Junction Tree Algorithms for Solving Sparse Linear Systems.* Technical Report UCB/CSD-03-1271, University of California, Berkeley. http://www.stanford.edu/~paskin/pubs/

[4] Graphical Models Reading Group, Presented by Chris Bartels, University of Washington Department of Electrical Engineering January 29, 2004 http://ssli.ee.washington.edu/~bilmes/grg/notes_Jan_29_2004.ppt

[5] Hans L. Bodlaender and Ton Kloks (1993), *Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs* Journal of Algorithms 21 (1996) 358-402.