A Fast Parallel Branch and Bound Algorithm for Treewidth

Yang Yuan
Department of Computer Science
Peking University
Beijing, China
Email: yangyuan@pku.edu.cn

***Abstract*—In this paper, we propose a Fast Parallel Branch and Bound algorithm (FPBB) for computing treewidth. FPBB searches the elimination ordering space using depth-first search approach, and employs multithreading techniques to search smaller divided spaces simultaneously. In order to protect the shared hash table without increasing the running time, we have designed a special algorithm for the readers-writers problem. We also present a new heuristic called similar group for refining search space. With other carefully chosen heuristics, FPBB performs well in computing both lower bound and upper bound on treewidth, and has solved 17 benchmark graphs whose exact treewidths were previously unknown. Additionally, FPBB is an anytime algorithm, which is insensitive to initial upper bound, and is able to find exact answer quickly. Computational results show that FPBB is significantly faster than the state-of-the-art algorithm proposed by Zhou and Hansen.**

I. INTRODUCTION

Introduced by [15], treewidth becomes an important notion used in many fields nowadays, including Constraint Satisfaction [12], Probabilistic Inference [13], Bucket Elimination [6], and Recursive Conditioning [5]. Since the complexity of these algorithms is exponential in the treewidth of the graph, a better elimination ordering will help improve the efficiency of these algorithms. Treewidth can also be regarded as a parameter to measure how much the graph resembles a tree. Researchers have computed the treewidth of java programs [10] and the Internet in order to get better understanding of their graph structures.

Moreover, many in general NP-complete problems can be solved in polynomial or even linear time, when the treewidth of the input graph is bounded. According to [14], if Strong Exponential Time Hypothesis [11] is true, current best known algorithms for a number of well-studied problems on graphs of bounded treewidth are essentially the best possible. Examples of these problems are INDEPENDENT SET, DOMINATING SET, MAX CUT, etc. Therefore, treewidth is important from both practical and theoretical perspectives.

However, it is NP-hard [1] to compute treewidth. Currently, on graphs with only 100 vertices, this problem remains difficult. Since physical constraints prevent frequency scaling and the power consumption of CPU has become a problem, parallel computing turns out to be the dominant paradigm in computer architecture in recent years, mainly in the form of multicore processors [2]. Following this trend, we have designed a Fast Parallel Branch and Bound algorithm (FPBB) for computing treewidth, which uses multithreading techniques and has better performance as it utilizes more cores. We use critical sections to protect shared data, and have designed a special algorithm for protecting the hash table, which allows duplicated states, but is much faster. To our knowledge, FPBB is the first practical parallel algorithm for treewidth.

FPBB is a depth-first search based anytime algorithm, which can find better solutions as it has more time to run. As we will show in Section IV, this is a big advantage for computing treewidth, because FPBB spends most time confirming that the upper bound found at the very beginning is the optimal answer. We also introduce a heuristic called similar group, which is time-efficient and helps reduce the search space. Moreover, we choose other heuristics carefully according to their cost-performance ratios. For example, we have discovered that Simplicial Vertex Rule has little effect on dense graphs.

We have solved 17 benchmark graphs and improved 12 known bounds of other graphs in TreewidthLIB. Empirical evaluation on benchmark graphs and random graphs shows FPBB is superior to the state-of-the-art algorithm proposed by [18].

II. RELATED WORK

The treewidth of a graph G is closely related to the elimination orderings. When we *eliminate* a vertex v in G, we make all the vertices that are adjacent to v form a clique by adding edges, and then remove v and all incident edges from G. An *elimination ordering* is a ordering of vertices for elimination. The *width* of an elimination ordering is the maximum degree of the vertices when they are eliminated from the graph. The *treewidth* of G equals to the minimum width over all possible elimination orderings. The new graph results from eliminating some vertices of G is called a *intermediate graph*.

In practice, there are many search algorithms for treewidth, such as QuickBB [9], BestTW [7] and the one combining BFS and DFS strategies (we refer to this algorithm as COMB in this paper) [18]. QuickBB is also a branch and bound algorithm, but being unaware of that the same set of vertices eliminated from a graph in any

order will result in the same intermediate graph, it needs to search the space of size $\Theta(n!)$, which grows too fast. BestTW is an improvement on QuickBB for it searches only $\Theta(2^n)$ nodes, and uses best-first search. However, in order to improve scalability, BestTW uses memory-efficient representations for intermediate graphs, which incur the overhead of intermediate graph generation every time one node is expanded.

COMB is the state-of-the-art algorithm, which further improves BestTW. Instead of using best-first search, COMB uses breadth-first search to save memory. However, like BestTW, COMB uses memory-efficient representations for intermediate graphs, which result in extra computation for generating them. The extra computation time is still considerable even though COMB introduces an efficient depth-first search approach to reduce the average generating time of intermediate graphs.

Moreover, COMB is not an anytime algorithm, that is, it needs to expand at least $n-k$ layers to obtain the answer if the treewidth of the graph is k, and it cannot update upper bound or lower bound on treewidth during its execution. This feature makes COMB very sensitive to the initial upper bound set before the search, which has great influence on the effect of refining search space.

Besides, because COMB only stores the search frontier in the memory, it is difficult for it to reconstruct the solution path after the goal is reached. According to [18], in order to find the elimination ordering, one should repeat running COMB many times using divide and conquer approach, which demands more time than simply computing treewidth.

III. ALGORITHM DESCRIPTION

A. Depth-First Search

FPBB searches the elimination ordering space using depth-first search approach. The search tree is constructed as follows. The root node is an empty set, and after expanding the root node, we will obtain n candidates for the first vertex in the ordering. After expanding any of these nodes, we will obtain $n-1$ candidates (for the first vertex is used) for the second vertex in the ordering, and the same step repeats until the n_{th} expansion, where there will be only one candidate left.

There are $\Theta(n!)$ nodes in the search tree, which are too many. In [4], it is proved that, the same set of vertices eliminated from the original graph in any order will produce the same intermediate graph. Thus, there are 2^n possible intermediate graphs in the search space. Fig. 1 is an example when n=4, in which each intermediate graph is represented by corresponding eliminated vertex set. We only need to search 2^n nodes, and duplicated nodes can be safely cut off.

A typical depth-first search algorithm will search the first child node of the root, and go deeper and deeper until it hits a node that has no children. Then the search backtracks, returning to the most recent node it has not finished exploring.

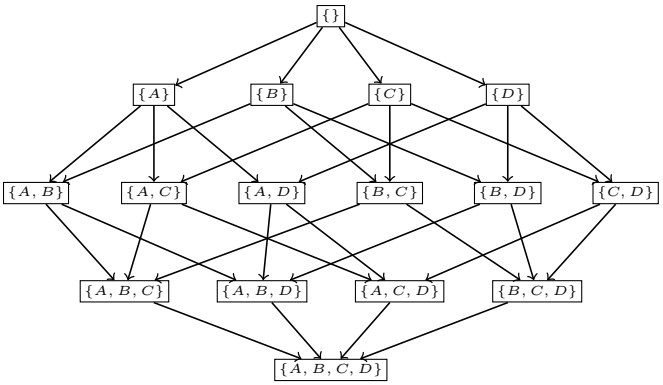

Figure 1. 2^n Intermediate Graphs in Search Space

FPBB improves the typical algorithm by using multi-threading techniques. Expanding nodes is a branching step; all the expanded nodes are independent. We allocate the expanded nodes to the free threads, and these threads can search the split subtrees simultaneously. We use a critical section [16] to protect the pool of free threads. After entering the critical section, current thread needs to check the number of free threads again in case the free threads have been used up. See Algorithm 1 for details.

Algorithm 1 Allocate expanded nodes to free threads

Input: $freeT$,$expNodes$

```
1: if expNodes > 1 & freeT.len > 0 then
2:     EnterCriticalSection(threadMutex)
3:         if freeT.len > 0 then
4:             useT ← Min(freeT.len, expNodes − 1)
5:             nodesPerT ← expNodes/ (useT + 1)
6:             pos ← 0
7:             for curT = 1 to useT do
8:                 pos ← pos + nodesPerT
9:                 Alloc(pos, freeT[curT], nodesPerT)
10:                StartThread(freeT[curT])
11:            end for
12:            expNodes ← nodesPerT
13:        end if
14:    LeaveCriticalSection(threadMutex)
15: end if
```

During the search, when a better upper bound is found, FPBB will update the upper bound. Another critical section is used for protecting current upper bound, in case more than one thread wants to modify it at the same time.

B. Hash Table for Detecting Duplicates

FPBB uses a hash table shared by every thread for duplicated nodes detection. The table stores reached states, and a parameter called LB is stored along with every state.

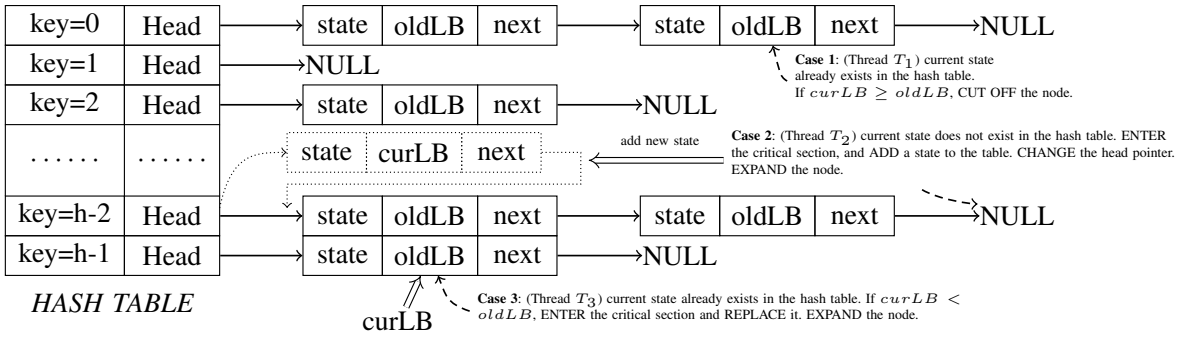

Figure 2. Different Cases for Expanding a New Node

LB means the lower bound of current searching elimination ordering, which is the maximum degree of the eliminated vertices so far. When duplicated state is detected, it should be replaced only if the LB of current state is not smaller than the LB saved in the hash table. Otherwise, the LB should be updated and current node should be expanded.

Since there are many threads operating at the same time, a critical section is used to ensure that the table can be safely accessed by many threads simultaneously. This is the classical readers-writers problem with the constraint that no process may access the hash table for reading or writing while another process is writing to it, and has standard solutions [16].

However, we use the hash table as a refining heuristic to cut off unnecessary branches. So there could be some duplicated states in the hash table, as long as the hash table does help reduce the search space in most cases. In order to get better performance, we put restrictions only for the writers: only one writer can access the table at a time, and any number of readers can access the table at any time. In this way, it takes less time for the readers to access the memory, without sacrificing the benefits of the writers. And we will show that it will not affect the result when there are conflicts between threads.

When one writer thread wants to modify the hash table, it will add a new state to the table, or improve the LB parameter of a state. We only discuss the first situation here, as the second one is similar. The hash table is implemented using separate chaining, so this thread can add the state to the head of the chain, and then change the head pointer of the table. If some reader threads want to access this chain at that time, they may fail to access the newly added state in the table for it is not completely added yet, but they can still access other states started from the head pointer. It is possible that some of their states are duplicates of the newly added state; since they have missed this state, they may add this state into the table again. But the chance is little and it will not affect the result if we add two or more identical states into the table.

The only negative result is that since there are possibly two or more identical states in the table, the access time of the readers may increase. As we will show in Section IV-C, the increased access time is negligible compared to the extra waiting time of the readers and writers in standard solutions. Fig. 2 gives an illustration for the whole process. See Algorithm 2 for the pseudo code. Every single statement in the pseudo code is assumed to be atomic.

Algorithm 2 Access hash table and add new states

Input: $curLB$,$curState$

```
1: key ← CalculateKey(curState)
2: block ← table[key]
3: update ← false
4: while block ≠ NULL do
5:     if SameState(curState, block.state) then
6:         if block.LB ≤ curLB then
7:             return
8:         else
9:             update ←true
10:            break
11:        end if
12:    else
13:        block ← block.next
14:    end if
15: end while
16: EnterCriticalSection(tableMutex)
17:    if update = true then
18:        if block.LB > curLB then
19:            block.LB ← curLB
20:        end if
21:    else
22:        tmp ←NewBlock(curState, LB)
23:        tmp.next ← table[key]
24:        table[key] ← tmp
25:    end if
26: LeaveCriticalSection(tableMutex)
```

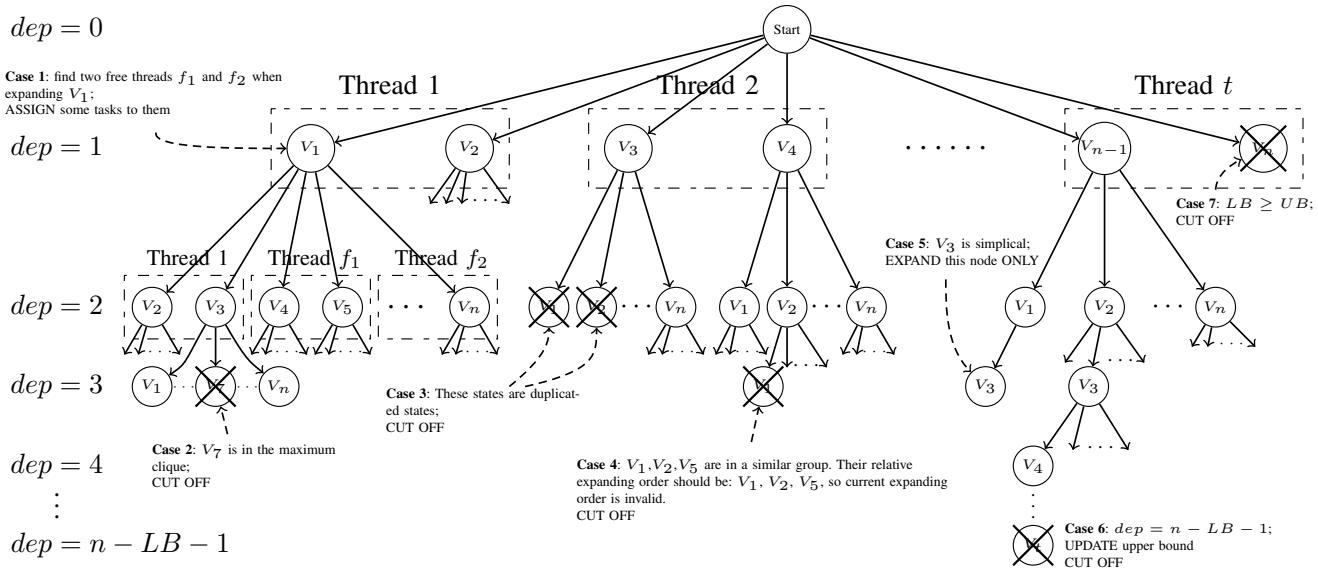

Figure 3. An Overview of FPBB

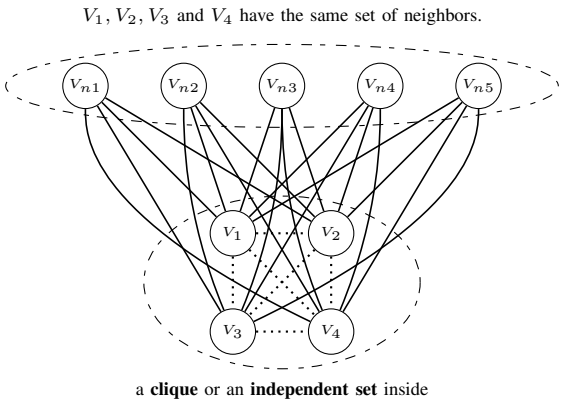

Figure 4. An Example for Similar Group

C. Heuristics for Refining

1) Similar Group: We introduce a new heuristic called similar group for refining search space. Two vertices v_0 and v_1 are *similar* if they share the same set of vertices that are adjacent to them. Note if v_0 and v_1 are connected, and they share all the other adjacent vertices, they are similar as well.

We will show that the similar property is an equivalence relation in the graph. Obviously, this property is reflexive and symmetric, so we only need to show that it is transitive, i.e., given three vertices v_0, v_1, v_2 which satisfy $v_0 \sim v_1, v_1 \sim v_2$, we have $v_0 \sim v_2$. If v_0 and v_1 are connected, since v_1 and v_2 share the same neighborhood, v_0 and v_2 are also connected. If v_0 and v_1 are not connected, v_0 and v_2 could not be connected, otherwise v_2 will have an adjacent vertex v_0, which v_1 does not share. For another vertex v_n, it is easy to see that v_n is either adjacent to all of these three vertices or none of them. Thus, $v_0 \sim v_2$ and the similar property is an equivalence relation. As shown in Fig. 4, all the vertices in a equivalence class form a clique or an independent set in the graph. This equivalence class is called a *similar group*.

The basic idea behind similar group is that the vertices in the same similar group are interchangeable in the elimination ordering. Given an elimination ordering $\pi = [v_{n1}, v_{n2}, \cdots, v_1, v_{nk_1}, b \cdots, v_2, v_{nk_2}, \cdots, v_3, \cdots]$, where v_1, v_2, v_3 are in a similar group, it is easy to see that the width of this ordering will not be affected after interchanging these three vertices' positions in the ordering. Actually, after swapping the labels of the vertices inside a similar group arbitrarily, we will get a same graph as the original one. So the changed ordering can be regarded as the "original ordering" for a changed graph with labels swapped.

Since the relative order of the vertices in a similar group does not affect the intermediate graphs produced, we can set a fixed order for them. Ideally, it will reduce the search space to $1/s!$ of the original size, where s is the size of the group. Sometimes there is more than one similar group in a graph, which may further reduce the search space.

Similar groups can be found very fast. Since all the similar groups can be regarded as disjoint subsets of the vertex set, we simply enumerate every pair of vertices v_i, v_j, and check whether they are similar or not. If they are similar, we can merge them into a similar group using disjoint sets data structure [17]. Thus, all similar groups in a graph G can be found in $\Theta(n^3)$ at the beginning of the search, where n is the number of vertices in G.

Similar groups can be used conveniently during the search. For all the vertices $v_1, v_2, v_3, \cdots, v_t$ in a similar group, we add a link between v_i and v_{i+1}, and maintain

a permission table for all the vertices. Initially, only v_1 has the permission to be eliminated, and when v_i is eliminated, we will find v_{i+1} through the link and give it the permission. Of course, v_{i+1} will lose its permission after backtracking to v_i during depth-first search. In this way, the maintaining cost is $O(1)$ for every expanding step.

2) Simplicial Vertex Rule: Reference [3] developed Simplicial Vertex Rule and Almost Simplicial Vertex Rule, which will not affect the treewidth of the graph. For a subset V_0 of the vertex set V, if every pair of vertices is connected in G, V_0 induces a clique. A vertex v is *simplicial* if its neighbourhood induces a clique. A vertex v is *almost simplicial* if the set of all but one of its neighbours induces a clique.

The *Simplicial Vertex Rule* states that if a vertex v_1 in a graph G is simplicial, there exists an elimination ordering starting with v_1 for G with exact treewidth. The *Almost Simplicial Vertex Rule* states that if a vertex v_1 of degree d in a graph G is almost simplicial and the lower bound on treewidth is at least d, there exists an elimination ordering starting with v_1 for G with exact treewidth. In other words, if simplicial or almost simplicial condition is satisfied, we only need to expand one node in the search tree, which greatly refines the search space.

The performance of this heuristic depends on graphs. On some graphs, this heuristic can help to reduce the search space by more than 90%, while on other graphs, it can hardly cut off any branches at all. Since it is time-consuming to test simplicial or almost simplicial conditions, it is unwise to apply this heuristic to all kinds of graphs. According to the experimental result in Section IV-B, we only apply it to sparse graphs.

3) Backtrack Condition: Since we maintain LB during the search, we can start backtracking earlier using a heuristic from [9]. If current intermediate graph has no more than $LB + 1$ vertices left, the parameter LB will not grow in the future elimination for this graph, because the maximum degree of its vertices is no more than LB. Thus, we will backtrack at that time.

4) Maximum Clique: Reference [4] proved that, for all cliques in the graph, there exists an elimination ordering with exact treewidth with the vertices of that clique as the last vertices of that ordering. So we can arrange the vertices in the maximum clique as last vertices in the ordering. Since treewidth is not smaller than the size of maximum clique of the graph, combined with the backtrack condition heuristic, we will never need to expand these vertices. Thus, during the search, we simply forbid these vertices to be expanded, which helps to cut off many branches.

We give an overview of FPBB in Fig. 3, including all heuristics we used in the algorithm.

D. Lower Bound and Upper Bound

For those graphs on which exact treewidth are difficult to compute, our algorithm is able to compute better upper bound and lower bound on treewidth quickly.

A lower bound LB means there is no elimination ordering whose width is less than LB. Thus, we set the initial upper bound for FPBB as LB, and FPBB will search all potential suitable elimination ordering in the search space. If the upper bound has not been updated after the search is complete, LB is indeed a lower bound; otherwise, we have found the exact treewidth.

We use a modified version of FPBB (MFPBB) to compute upper bound. Rather than fully expanding currently searching node, MFPBB expands only three most promising child nodes at a time, which greatly reduces the search space. We use *minor-min-width* heuristic [9] to compute a lower bound for each child node, and child nodes with small lower bound are considered promising. When two child nodes have the same lower bound, we pick the one with fewer edges, because sparse graphs usually have smaller treewidth. MFPBB also has constraints on the total number of nodes expanded. If it has expanded more than 30000 nodes, it will output current upper bound and halt.

Additionally, since FPBB is an anytime algorithm, it will continue to give better upper bound before it find the exact answer. In Section IV-A, we use FPBB to compute better upper bound on benchmark graphs.

IV. EXPERIMENTAL RESULTS

We have implemented FPBB and COMB in C++. All the benchmark graphs we used come from TreewidthLIB[1]. The experiments were conducted with Intel Core i7-870 Processor and 8 GB RAM on Windows 7. Since the processor has 4 cores, we use 4 threads in FPBB.

Before the search, we will preprocess the graphs. All the vertices will be sorted according to their degrees; vertices with large degrees will be put in front of those with small degrees. Thus, FPBB will try to eliminate vertices with large degrees first, and better upper bound are easier to be obtained in this way, as discussed in [18].

A. Benchmark Graphs

In this experiment, we try to improve both the upper bound and the lower bound of the benchmark graphs using FPBB. For a graph, if its upper bound meets the lower bound, it is *solved*. FPBB has solved 17 benchmark graphs whose exact treewidths were previously unknown, and improved 12 known bounds of the others. Due to limited space, we only list some of our results in Table I. In the table, improved bounds are emphasized using bold font. Columns show the name of the graph (Name), the number of vertices (N), known upper bound (UB) and lower bound (LB) on

[1] http://people.cs.uu.nl/hansb/treewidthlib/index.php

Table I
IMPROVED BOUNDS OF BENCHMARK GRAPHS.

Name	N	UB	LB	Exp	Sec
1dj7	73	**26**	26	1255	0.02
1ku3	60	**22**	22	1587	0.02
1kw4	67	**27**	27	342097	1.40
1i27	73	**26**	26	443639	2.15
1c9o	66	**28**	28	907015	3.84
1e0b	59	24	**24**	7455969	20.84
1r69	63	**29**	29	6376178	21.31
miles750	128	36	**36**	5703212	22.59
1en2	69	**16**	16	10676259	50.97
1qtn	86	**23**	23	10181450	72.67
1gcq	67	30	**30**	132623386	645.50
queen9_9*	81	58	**53**	250602432	758.95
1d3b	68	25	**25**	209300902	1328.91
1c4q	67	31	**29**	297316678	1808.00
1l9l	70	**28**	28	333484641	1844.53
1cc8	70	32	**29**	350271892	2854.10
1fse	67	**26**	25	282549	1.56
		26	**26**	108955514	706.78
1hj7	66	**28**	27	8864657	38.81
		28	**28**	227279244	1180.41
1dp7	76	**26**	25	955153	4.85
		26	**26**	294211812	2176.82

treewidth, number of expanded nodes (Exp), and running time in seconds (Sec). For last 3 graphs, FPBB improves both upper bound and lower bound on treewidths of them. A "*" indicates that the Simplicial Vertex Rule is not applied.

From the table, we can see that FPBB is able to find better upper bounds of some graphs quickly. For those graphs whose exact treewidths are still unknown, such as *queen9_9*, FPBB is able to improve its lower bound and shorten the gap between the upper bound and the lower bound.

B. Random Graphs

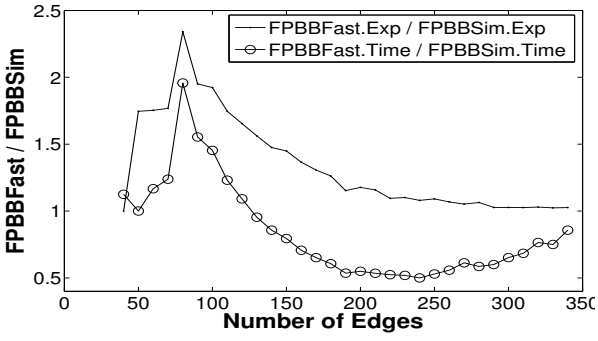

Figure 5. A Comparison of Expanded Nodes and Runtimes Between FPBBSim and FPBBFast

In this subsection, we run FPBB on random graphs generated in $G(n, m)$ Model [8].

Firstly, we generate random graphs with $n = 35$ and different m to show the effect of the Simplicial Vertex Rule. We run both FPBB with Simplicial Vertex Rule (FPBBSim) and FPBB without it (FPBBFast) on these graphs, and compare the numbers of their expanded nodes. We enumerate m from 40 to 340 with step of 10. For each m, we generate 100 random graphs and compute the average expanded nodes of two variations. Fig. 5 gives a comparison between FPBBSim and FPBBFast. It is clear that Simplicial Vertex Rule works well when the graph is sparse.

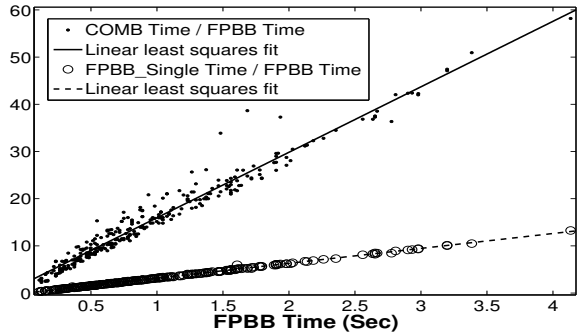

Figure 6. Ratio of COMB and FPBB_Single Runtime to FPBB Runtime on 300 Random Graphs

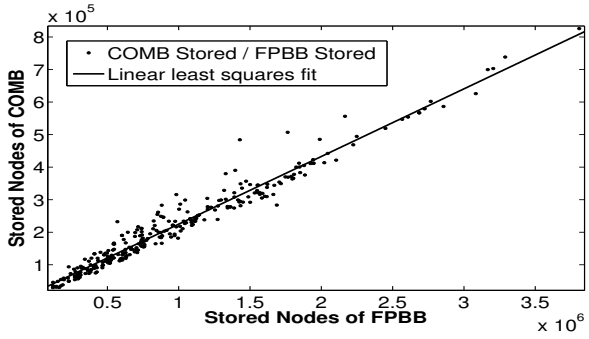

Figure 7. Ratio of COMB Stored Nodes to FPBB Stored Nodes on 300 Random Graphs

In the second experiment, we generate 300 random graphs with the $n = 35$ and $m = 140$, a parameter set used in both [7] and [18]. We run COMB, FPBB and FPBB_Single (FPBB with single thread implementation) on these graphs, and compare the running times of them in Fig. 6. Lines in the figure show linear least squares fit of data. Since the running time of COMB will be greatly influenced by the initial upper bound, we set the initial upper bound as the exact treewidth of the graph to obtain steady results in this experiment. So the results of COMB in this experiment are probably improved. On average, FPBB is about 14 times faster than COMB, and 3.1 times faster than single thread implementation. Since we use the CPU with only 4 cores, and the shared hash table is accessed quite heavily during the execution (every time one thread expands a new node, it needs to access the hash table), we believe it is a satisfying result.

Table II
A COMPARISON AMONG FPBB, ITS VARIATIONS AND COMB

			FPBB				UB+5		Variations(Sec)			COMB	
Name	TW	UB	Alo	LUT	Exp	Sec	LUT	Sec	ST	RF	WF	Exp	Sec
queen6_6*	25	25	7	0.03	11187	0.03	0.02	0.03	0.06	0.18	0.16	11185	0.39
miles500	22	22	0	0.03	2	0.03	0.03	0.03	0.05	0.05	0.05	2	0.02
inithx.i.1	56	56	3	0.09	1392	0.09	0.08	0.09	0.08	0.09	0.09	1305	15.07
queen7_7*	35	35	18	0.02	533999	0.67	0.02	0.66	1.79	11.06	9.64	529242	26.96
myciel5	19	19	18	0.03	3482899	4.71	0.03	4.71	14.02	33.74	32.18	3351675	73.20
miles750	**36**	38	24	1.39	5703212	22.59	3.01	26.08	79.54	34.82	38.44	-	-
queen8_8*	45	46	24	16.10	18345352	33.73	16.15	33.87	107.62	441.22	390.63	-	-

FPBB stores all the intermediate graphs produced so far, while COMB only stores the intermediate graphs in current layer. However, COMB needs to maintain a FIFO queue to perform breadth-first search, which demands additional memory. Besides, since the number of intermediate graphs in each layer grows exponentially as the depth increases, the whole search space is only several times larger than the size of largest layer. In this experiment, FPBB stores about 4 times more nodes than COMB on average. See Fig. 7. However, in most cases, if the exact treewidth is not known before the search, COMB will expand exponentially more nodes (thus requiring exponentially larger memory) than FPBB.

C. DIMACS GRAPHS

In this experiment, we compare FPBB with its variations and COMB using DIMACS Vertex Coloring Graphs in Table II. Columns show the name of the graph (Name), exact treewidth(TW), initial upper bound found by MFPBB before the search (UB), times of allocating expanded nodes to free threads (Alo), Last Updated Time for upper bound in seconds, i.e., time used for finding the elimination ordering of the exact treewidth (LUT), number of Expanded nodes (Exp), running time in Seconds of FPBB (Sec), Single Thread implementation (ST), Readers-Preference algorithm (RF), and Writers-Preference algorithm (WF). The big column UB+5 means that we run FPBB with initial upper bound 5 larger than TW in order to test its sensitivity to the initial upper bound. We also list the number of expanded nodes and running time of COMB in the last column.

According to the results, our implemented version of COMB is about 2 to 6 times faster than COMB implemented in [18], and has less expanded nodes. It shows that our implementation of COMB is a good one, which makes the comparison between COMB and FPBB seem reasonable.

The upper bound found by MFPBB is very close to the exact treewidth. The Allocated times are relative small compared to the expanded nodes, which means there are little time wasting on allocating new tasks to free threads during the execution. And the last updated time is surprisingly short, which means FPBB spends most time confirming the upper bound it found at the very beginning is the optimal answer.

Comparing to COMB, FPBB is significantly faster. On the graph *queen7_7*, FPBB is about 40 times faster than COMB. This gap is even larger than the gap on random graphs, which indicates that FPBB may have better performance on difficult graphs. Since we use the same heuristics in COMB, the node expansions of the two algorithms are similar. But comparing to [18], the node expansion is improved. For example, on the graph *queen7_7*, expanded nodes of FPBB are only 57.1% of expanded nodes of COMB in [18]. FPBB has significantly less average processing time for each expanded node than COMB, because it does not need to generate the intermediate graph or maintain a FIFO queue. Moreover, it is easy for FPBB to save the corresponding elimination ordering when updating the upper bound, while COMB needs to use divide and conquer approach to reconstruct the solution path.

We also implemented different variations of FPBB, such as single thread implementation, and two standard solutions to the readers-writers problem stated in [16]. If we use the *readers-preference* algorithm, the hash table cannot be updated in time, and duplicated states are more likely to be added into the table. If we use the *writers-preference* algorithm, all other threads need to wait if one thread is adding a new state. Besides, these two algorithms require more semaphores, which bring additional waiting time.

From the table, we can see that they are even slower than single thread implementation on most graphs. A special case is *miles750*, in which ST takes more time. It is because the optimal elimination ordering starts with a vertex with small degree, and can be found quickly by threads after allocating nodes. But if there is only one thread, it needs to do much computation before searching this vertex, and the initial upper bound cannot be updated in time, which results in larger searching space. This is also the reason why COMB fails to compute treewidth on last two graphs: the initial upper bound is not the exact treewidth, and cannot be updated during the execution, so the search space is much larger than the search space of FPBB.

From the UB+5 column, we can see that FPBB is not

sensitive to the initial upper bound. Both last updated time and total running time are not significantly affected after the initial upper bound is enlarged by 5.

V. CONCLUSIONS

In this paper, we have presented a fast parallel branch and bound algorithm for computing treewidth, which is based on depth-first search in the elimination ordering space. We use a modified version of standard solution to the readers-writers problem to get better performance. Benefited from depth-first search approach, FPBB is an anytime algorithm, which makes it insensitive to the upper bound initially set. We have proposed a new heuristic called similar group for refining search space. Combined with other carefully chosen heuristics, FPBB has solved 17 benchmark graphs and improved 12 known bounds of other graphs. Experimental results show that FPBB is much faster than state-of-the-art algorithm COMB and is good at computing lower bound and upper bound on treewidth.

For future work, we want to further investigate the effect of different heuristics on graphs with different features, which may help we choose the heuristics according to the situation. Moreover, since the basic idea behind similar group is to use graph isomorphism to cut off branches, we believe this idea can be further developed to get stronger heuristic.

ACKNOWLEDGEMENTS

The author would like to thank Chen Wang, Zining Zheng and Shuyang Gao for helpful discussions. Moreover, he wants to thank three anonymous reviewers for their advice which greatly improves an earlier version of this paper.

References

[1] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Alg. and Discrete Methods*, 8:277–284, 1987.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] H. L. Bodlaender, A. M. C. A. Koster, and F. van den Eijkhof. Preprocessing rules for triangulation of probabilistic networks. *Computational Intelligence*, 21(3): 286–305, 2005.

[4] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. In *Algorithms - ESA 2006, 14th Annual European Symposium, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 672–683. Springer, 2006.

[5] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1–2):5–41, 2001.

[6] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[7] P. A. Dow and R. E. Korf. Best-first search for treewidth. In *AAAI 2007*, pages 1146–1151, 2007.

[8] P. Erdős and A. Rényi. On random graphs. I. *Publ. Math. Debrecen*, 6:290–297, 1959.

[9] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence, UAI 2004*, pages 201–208, 2004.

[10] Gustedt, Maehle, and Telle. The treewidth of java programs. In *ALENEX: International Workshop on Algorithm Engineering and Experimentation, LNCS*, 2002.

[11] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *J. Comput. Syst. Sci*, 62(2):367–375, 2001.

[12] A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40(3):170–180, 2002.

[13] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Statis. Soc. B*, 50:157–224, 1988.

[14] D. Lokshtanov, D. Marx, and S. Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. *CoRR*, 2010.

[15] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, Sept. 1986.

[16] W. Stallings. *Operating Systems: Internals and Design Principles, Sixth Edition*. Prentice Hall, 2008.

[17] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):448–501, Apr. 1975.

[18] R. Zhou and E. A. Hansen. Combining breadth-first and depth-first strategies in searching for treewidth. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 640–645, 2009.