Volume 4, Issue 3, March 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Faster Branch and Bound Algorithms for Solving the Maximum Clique Problem

Kavita Gupta*, K K Shukla
Department of Computer Engineering
Indian Institute of Technology
(Banaras Hindu University), India

Abstract— We present efficient branch and bound algorithms for solving the Maximum Clique Problem. First strategy is to store the size of clique consisting of a seed vertex in a subset induced by candidate set of the seed vertex. Results stored are further used to reduce the computation involved in computing clique consisting of other seed vertices in another subset of the graph. This approach along with greedy algorithm for graph coloring, reduce the search space but does not always outperform the first strategy due to overhead involved in computing the graph coloring. Experiments on DIMACS benchmarks and random graphs demonstrate that the improved branch and bound algorithms are faster than the original algorithm by Carraghan and Pardalos [7].

Keywords— Clique, Branch and Bound Algorithm, Maximum Clique Problem, Greedy algorithm, Graph Coloring

I. INTRODUCTION

Maximum Clique Problem [1] is known to be NP-Complete and consists of finding the size of maximum possible clique in an undirected graph. A clique in an undirected graph can be defined as a set of vertices such that there exists a path between every two pair of vertices of the graph. In other words, a clique is a complete sub-graph of a given graph. Consider any graph G(V, E) where V is the set of *n* vertices and E is the set of *m* edges. In any such graph we can define clique as the set of vertices V' and a set of edges E' such that V' is a subset of V and E' is a subset of E and there exists an edge between every pair of vertices in V'. Clearly, there can be various possible cliques in any given graph. The problem of Maximum Clique aims to find out a clique which covers the maximum number of vertices. The size of maximum clique is known as clique number and is denoted by □.

A. Literature Review

Maximum Clique Problem has been used in various fields of computer science, operational research, etc. Cliques were enumerated by Balas and Yu [5]. Their idea was to search for a maximal induced triangular sub graph T of G in which a maximum clique C is searched. Further a branch and bound based technique was introduced by Wood [3]. Caprara, Pisinger and Toth [2] proposed a solver for the quadratic Knapsack problem which was applied to the maximum clique problem. Cuts have also been used for solving maximum clique problem using integral programming. Carraghan and Pardalos [7] have proposed a branch and bound algorithm for solving the problem which is discussed in Section I.B. Further, the idea of using graph coloring and a new strategy to cut down the computational effort using an extra storage and the details of our approach is given in section II. DIMACS instances have been used as a benchmark for testing the performance of the proposed algorithm. We have reported experimental results in Sec III.

B. Branch and Bound Algorithm for Maximum Clique

Branch and Bound [7] is an optimization algorithm, in which all the candidate solutions are enumerated and the solution set is reduced using a lower and upper bound. This bound is computed on the basis of the quantity which is to be optimized. In Maximum Clique Problem, the branching is done through enumerating all possible solutions as in Line 5 of the recursive algorithm shown in section I.C. In order to place a bound on the candidate set, the neighborhood vertices are considered in the candidate set and all other vertices are ignored since they would not result in the maximum clique. This can be seen in Line 7 of the algorithm in section I.C.

C. Algorithm

Branch_and_Bound *(clique, candidate)*

1. **if** *(|clique|>|maxclique|)* **then**
2. maxclique=clique
3. **end if**

4. **if** *(|clique|+|candidate|>|maxclique|)* **then**
5. **for all** *vertices v* *candidate*
6. *newclique=clique* *v*
7. *newcandidate=candidate* *N(v)*
8. **Branch_and_Bound** *(newclique , newcandidate)*
9. **end for**
10. **end if**

Main *()*

1. **Branch_and_Bound** *(NULL, V)*

In the above pseudo code the symbols used have following meaning:
V: set consisting of all vertices
clique: current clique set
candidate: current candidate set
maxclique: global maximum clique set
N(v): is the neighboring vertices of vertex v in the candidate set
newclique: new clique set formed by including the current vertex(v) in the current clique set
newcandidate: new candidate set formed by including vertices present in current candidate set and present in neighborhood of current vertex(v)

D. Complexity

Branch and Bound [7] is a recursive algorithm which enumerates all the possible candidate solutions. Consider a graph with *n* vertices. In the worst case it can be completely connected which means the graph is itself the maximum clique. In such a case, there will be 2^n cliques. Therefore, it creates a worst case complexity of $O(2^n)$ for finding the maximum clique by candidate generation.

II. IMPROVEMENTS IN BRANCH AND BOUND ALGORITHM

In the algorithm proposed by Carraghan and Pardalos [7], NULL set is considered as the current clique and vertices are added recursively in order to obtain the maximum clique. We have discussed two strategies in section 2.A and section 2.B to improve this algorithm.

A. Strategy 1: Using an extra storage

In this strategy, the maximum clique containing a seed vertex v_i Î V over the candidate set N' (v_i , {v_1, $v_{2, \dots ,}$ v_i}) is computed. Here, the notation N'(v_i ,{v_1,$v_{2, \dots ,}$ v_i}) represents the set of neighboring vertices of v_i in set {v_1, $v_{2, \dots ,}$ v_i}. It excludes the vertices {v_{i+1},… v_n}. This result is stored as dp[v_i]. Therefore, dp[v_i] is size of the maximum clique necessarily containing vertex v_i and some vertices from the candidate set N'(v_i ,{v_1, $v_{2, \dots ,}$ v_i}). Each iteration considers vertex v_i V as the seed vertex. We have considered the selection of seed vertices in lexicographic order. So, when computing dp[v_j] such that $v_j > v_i$, the vertex v_i is included only if it satisfies the following relation:

$$|\text{clique}| + dp[v_i] > |\text{maxclique}|$$

If a vertex v_i does not satisfy the above relation, then it is discarded from branching. Therefore, the computations on each iteration are reduced using the result dp stored in previous iterations. Using this approach we have reduced the computations by using extra space. We have used an auxiliary array of size *n* for storing the intermediate results. The improved algorithm using the above stated strategy is as follows:

Fast_Branch_and_Bound *(clique, candidate)*

1. **if** *(|clique|>|maxclique|)* **then**
2. *maxclique=clique*
3. **end if**
4. **if** *(|clique|+|candidate|>|maxclique|)* **then**
5. **for all** *vertices v* *candidate* **do**
6. **if** *(|clique|+dp[v]>|maxclique|)* **then**
7. *newclique=clique* *v*
8. *newcandidate=candidate* *N(v, candidate)*

9. **Fast_Branch_and_Bound** *(newclique , newcandidate)*
10. **end if**
11. **end for**
12. **end if**

Main ()

1. *maxclique={v_1}*
2. **for all** *vertices* $v_i \in V$ *in lexicographic order* **do**
3. *V'=N (v_i ,{v_1, $v_{2,\ldots,}v_i$})*
4. **Fast_Branch_and_Bound** *(v_i, V')*
5. *dp[v_i] = |maxclique|*
6. **end for**

In the above pseudo code:
clique: current clique set
candidate: current candidate set
maxclique: global maximum clique set. Initially set to NULL (global)
N'(v, S): neighboring vertices of vertex v in set S
newclique: new clique set formed by including the current vertex(v) in the current clique set
newcandidate: new candidate set formed by including vertices present in current candidate set and present in neighborhood of current vertex(v)
dp: auxiliary array of *n* elements such that each element corresponds to a node.
Although the asymptotic time complexity of the new algorithm using storage based strategy remains exponential, in practice it runs considerably faster on real benchmark problems and requires O(*n*) extra space.

B. Strategy 2: Using a greedy graph coloring

While computing the maximum clique, the search space can reduced by greedy coloring of candidate set in Line 4 of algorithm Fast_Branch_and_Bound in section 2.A. This is beneficial since the number of colors found using the function *GreedyColors* over the candidate set forms an upper bound on the size of maximum clique present in the candidate set, thus reducing the search space. The algorithm for greedy coloring is as shown:

GreedyColors (*candidate*)

1. **for each** *vertices* $v \in$ *candidate* **do**
2. *Color[v] = 0*
3. **end for**
4. **for** *i = 1 to n*
5. *AssignedColor[i] =* **false**
6. **end for**
7. **if** *|candidate| = 0* **then**
8. **return** *0*
9. **end if**
10. *Color [v_1] = 1*
11. *AssignedColor[1] =* **true**
12. **for** *i = 2 to |candidate|* **do**
13. *Color [v_i] =* **min (** *k [1,i] | Color [w] != k* $\forall$ *w* $\in$ *N(v_i, {v_1,v_2,…,v_i})* **)**
14. *AssignedColor [Color [v_i]] =* **true**
15. **end for**
16. *Count = 0*
17. **for** *i = 1 to n* **do**
18. **if** *AssignedColor [i] =* **true then**

19. *Count = Count+1*

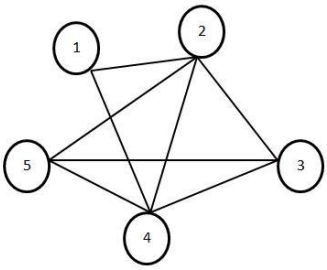
20. **end for**
21. **return** *Count*

The above function returns the number of colors required to color the graph represented by the candidate set. Here,
Color: Array of |candidate| elements representing the color assigned to each vertex in the candidate set.
AssignedColor: Boolean array of *n* colors where true means that the color has been assigned and false means that the color is available.
Greedy coloring gives an upper bound on the clique size which results in the reduced search space but it involve an overhead which may increase the CPU-time of the algorithm.

C. Example

The working of our algorithm is explained with a small undirected graph:

FIGURE 1 - EXAMPLE GRAPH



While computing the maximum clique using Carraghan's Branch and Bound algorithm (*algo-1*), following table is generated. There are **13 iterations** of the function Branch_and_Bound. The entries marked in bold in Table-1 are recursion termination points where the algorithm ceases calling itself. These occur only when:
|clique|+|candidate|<=|maxclique|.

TABLE 1 -ITERATIONS OF CARRAGHAN'S BRANCH AND BOUND ALGORITHM ON THE EXAMPLE GRAPH OF FIGURE 1

Iteration Number	**Clique**	**Candidate**	**|maxclique|**
1	NULL	{1,2,3,4,5}	0
2	{1}	{2,4}	1
3	{1,2}	{4}	2
4	**{1,2,4}**	**NULL**	**3**
5	{2}	{1,3,4,5}	3
6	**{1,2}**	**{4}**	**3**
7	{3}	{2,4,5}	3
8	{2,3}	{4,5}	3
9	{2,3,4}	{5}	3
10	**{2,3,4,5}**	**NULL**	**4**
11	{4}	{1,2,3,5}	4
12	**{1,4}**	**{2}**	**4**
13	**{5}**	**{2,3,4}**	**4**

Further on iterating through our improved algorithm without greedy coloring (*algo-2*), the similar table is generated which shows that there are **11 iterations** of the function Fast_Branch_and_Bound.

TABLE 2- ITERATIONS OF FAST BRANCH AND BOUND WITHOUT GREEDY COLORING ON THE EXAMPLE GRAPH OF FIGURE 1

Number	**i**	**Clique**	**Candidate**	**|maxclique|**	**dp[i]**
1	1	**{1}**	**NULL**	**1**	**1**
2	2	{2}	{1}	1	-
3		**{1,2}**	**NULL**	**2**	**2**
4	3	**{3}**	**{2}**	**2**	**2**
5	4	{4}	{1,2,3}	2	-

6		{2,4}	{1,3}	2	-
7		**{1,2,4}**	**NULL**	**3**	**3**
8	5	{5}	{2,3,4}	3	-
9		{4,5}	{2,3}	3	-
10		{2,4,5}	{3}	3	-
11		**{2,3,4,5}**	**NULL**	**4**	**4**

The entries marked in bold in Table-2 are recursion termination points where the algorithm ceases calling itself. These recursion termination points occur in either of the following conditions:

1. |clique|+|candidate|<=|maxclique| , or
2. |clique|+dp[v_i]<=|maxclique|

This shows that we require fewer computations for finding maximum clique with the improved algorithm by using extra space. The advantage is significant on larger instances.

Further on iterating through our improved algorithm with greedy coloring (*algo-3*), the similar table is generated which shows that there are **12 iterations** of the function Fast_Branch_and_Bound.

TABLE 3 - ITERATIONS OF FAST BRANCH AND BOUND WITH GREEDY COLORING ON THE EXAMPLE GRAPH OF FIGURE 1

Number	**i**	**Clique**	**Candidate**	**|maxclique|**	**dp[i]**
1	1	**{1}**	**NULL**	**1**	**1**
2	2	{2}	{1}	1	-
3		**{1,2}**	**NULL**	**2**	**2**
4	3	{3}	{2}	**2**	**-**
5		**{2,3}**	**NULL**	**2**	**2**
6	4	{4}	{1,2,3}	2	-
7		{2,4}	{1,3}	2	-
8		**{1,2,4}**	**NULL**	**3**	**3**
9	5	{5}	{2,3,4}	3	-
10		{4,5}	{2,3}	3	-
11		{2,4,5}	{3}	3	-
12		**{2,3,4,5}**	**NULL**	**4**	**4**

The entries marked in bold in Table 3 are recursion termination points where the algorithm ceases calling itself. These recursion termination points occur in either of the following conditions:

1. |clique|+GreedyColors (candidate) <=|maxclique| , or
2. |clique|+dp[v_i] <=|maxclique|

Although this shows that *algo-2* requires less iteration than *algo-3* and *algo-3* requires less iterations than *algo-1* , after experimenting on various DIMACS and random graphs we found out that both the algorithms (*algo-2* and *algo-3*) outperforms *algo-1* but no proper conclusion can be drawn on comparison of performance of both the algorithms (*algo-2* and *algo-3*). Since there is an overhead involved in *algo-3* it may result in greater CPU time than *algo-2*.

III. IMPLEMENTATION

The original branch and bound algorithm considers the candidate set to be all the neighboring vertices of the current clique vertices whereas the improved branch and bound algorithm considers a particular vertex v_i and finds the maximum possible clique in the candidate set (which is neighboring vertices of v_i in $\{v_1, v_2, \ldots, v_i\}$) for v_i. This result can be used to compute maximum cliques containing any vertex v_k where $v_k > v_i$ thus reducing the computations. Experiments on graph coloring show that it results in reduced search space. All the algorithms were implemented and compiled using GNU C++ compiler on Intel® Core™ i3 CPU M 350 @ 2.27GHz × 4 processor. The Operating System used was 64-bit 4:4.7.2-1ubuntu2.

A. Benchmark Problems

The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS)[4] is collaboration between Rutgers University, Princeton University, and the research firms AT&T, Bell Labs, Applied Communication Sciences, and NEC. DIMACS is devoted to both theoretical development and practical applications of discrete mathematics and theoretical computer science. It encourages, inspires, and facilitates researchers in these subject areas, and sponsors conferences and workshops. DIMACS instances have been used as benchmarks for comparative performance analysis of our algorithm against the traditional Branch and Bound Algorithm. DIMACS test cases used are given below:

1) Kel: Problems based on Keller's conjecture on tilings using hyper cubes.
2) Brock: Random graphs with cliques hidden among nodes that have a relatively low degree

3) Ham: A Hamming graph with parameters n and d has a node for each binary vector of length n. Two nodes are said to be adjacent if and only if the corresponding bit vectors are hamming distance at least d apart.
4) Cfat: Problems based on fault diagnosis problems

B. Random Graphs

We used the graph data generator gengraph_win to generate graph datasets in which the degrees of the nodes obey power law distribution. More specifically, gengraph_win generates a degree sequence which is composed by a set of integers according to several parameters assigned by user. The parameters include the count of degree numbers *n* , the minimum degree *min*, the maximum degree *max*, the average degree *avg*, and the exponent of the power law distribution α.

In power law topology, the frequency f_d of a degree is proportional to the degree d(>=2) raised to the power of a constant α i.e. $f_d \propto d^{\alpha}$

Note: On increasing the value of α, the number of nodes of large degree decreases
In our experiment, we have generated random graphs of a particular size *n* such that the exponent α = -2.5, *min*=1, *max*=*n*, and *avg* varies to generate graphs of varying density such that *avg* [*min*, (*min* + *max*)/2].

C. Profiler used for measuring computational time

Gprof[6] is a performance analysis profiler for UNIX applications. It uses a combination of instrumentation and sampling. Instrumentation code is automatically inserted into the program code during compilation (for example, by using the '-pg' option of the g++ compiler), to gather caller-function data. A call to the monitor function 'mcount' is inserted before each function call. Sampling data is saved in 'gmon.out' file just before the program exits, and can be analyzed with the 'gprof' command-line tool. The output consists of two parts which are as follows:

1) The flat profile which gives the total execution time spent in each function and its percentage of the total running time. It also reports the function call counts. Output is sorted by percentage, with hot spots at the top of the list.
2) The textual call graph, which shows for each function who called it (parent) and who it called (child subroutines).

The performance testing results are summarized in the following table. Here, the value of density of an undirected graph is computed as follows:
Density = number of edges/maximum edges possible = (edges*2)/ (n*(n-1)) Here,
d= density
 = clique number (size of the maximum clique)
algo-1= Carraghan's branch and bound algorithm (1990)
algo-2= improved algorithm without greedy coloring
algo-3= improved algorithm with greedy coloring
Branches = number of times the recursive Branch_and_Bound (or Fast_Branch_and_Bound) function is called.

TABLE 4 - EXPERIMENT ON DIMACS INSTANCES

S.No.	DIMACS instance	*n*	*d*		Branches			CPU time(sec)		
					Algo-1	***Algo-2***	***Algo-3***	***Algo-1***	***Algo-2***	***Alog-3***
1	keller4	171	0.64	11	23539510	1395636	87124	72.3	4.54	2.13
2.a	brock200_2	200	0.5	12	1170054	125990	14724	5.04	0.55	0.54
2.b	brock200_4	200	0.66	17	73845504	12530243	818840	303.08	52.18	26.89
3.a	hamming8-4	256	0.64	16	175493740	2345	2273	753.72	0.02	0.3
3.b	hamming6-2	64	0.9	32	3130173	626	595	3.44	0	0.01
4	cfat500-1	500	0.04	14	4105	1077	1000	0.03	0.01	0.94

D. Conclusion and future work

We conclude that-

1) For a particular graph, the number of branches varies as shown in Table-5. It is clear that the number of branches of *algo-1 >algo-2>algo-3* which is obvious since *algo-2* reduce the branching using an extra storage and *algo-3* further reduce the search space by graph coloring.

TABLE 5 - EXPERIMENT ON RANDOM GRAPH (N=100)

d	□	**Branches**			**CPU time (sec)**		
		algo-1	***algo-3***	***algo-2***	***algo-1***	***algo-3***	***algo-2***
0.05	6	223	125	128	0	0	0
0.1	9	526	137	174	0	0	0
0.15	12	6148	256	335	0.01	0	0
0.2	12	3717	295	664	0	0	0
0.25	20	44935	291	726	0.08	0	0
0.3	20	139882	2108	3330	0.26	0.04	0
0.35	24	212652	4998	6934	0.36	0.01	0.01
0.4	24	179162	10749	13687	0.32	0.25	0.02
0.45	25	206761	37252	50103	0.56	0.37	0.1
0.6	15	95375	8545	26090	0.19	0.13	0.06
0.75	35	553952	5226	15537	0.94	0.14	0.04
0.85	27	33746916	223811	6126727	59.49	6.42	12.04

2) As can be seen from Table-5 CPU-time for *algo-1* is always greater than CPU time for *algo-2* and *algo-3*. No proper conclusion can be made of the comparison of performance of *algo-2* and *algo-3*. In some cases *algo-2* outperforms (for example, n=100, d=0.4) whereas in some cases *algo-3* outperforms (for example, n=100, d=0.85). The reason is the overhead involved in *algo-3* in computing the graph coloring in order to prune the search space.

A similar experiment is performed on random graphs of n =200 by varying density as shown in Table-6.

TABLE 6 - EXPERIMENT OF RANDOM GRAPH (N=200)

d	□	Branches			CPU time (sec)		
		Algo-1	*Algo-2*	*Algo-3*	*Algo-1*	*Algo-2*	*Algo-3*
0.025	6	472	231	220	0	0	0
0.05	9	1224	379	231	0	0	0
0.07	14	7951	458	412	0.03	0	0
0.1	17	44710	735	458	0.16	0	0.03
0.15	20	154919	4401	3524	0.57	0.02	0.11
0.2	24	72295	3655	1058	0.40	0.01	0.03
0.3	45	106320562	20862	15070	43.51	0.08	1.16

3. As can be seen from Figure-2, *algo-1* underperforms *algo-2* and *algo-3*. Figure-2 is a plot of various DIMACS instances v/s CPU-time. For-example keller4 is a DIMACS instance of n=171 and the CPU-time can be seen in Table-4.

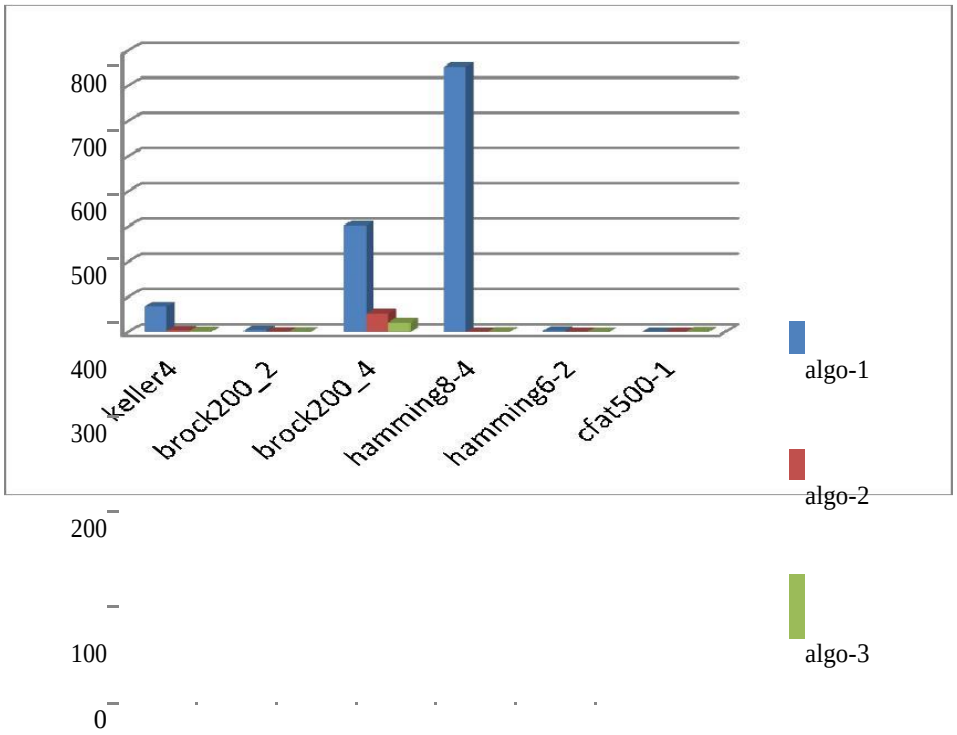


FIGURE 2- DIMACS INSTANCES V/S CPU-TIME

1. Speed up increases on increasing the density in the fashion as shown in figure-3 and figure-4 for *algo-1* and *algo-2*, respectively on random graphs of n=100 and n=200. As can be seen from Figure-3 and Figuire-4, speedup increases with increase in density and increase in speed up is faster for n=200 than n=100.

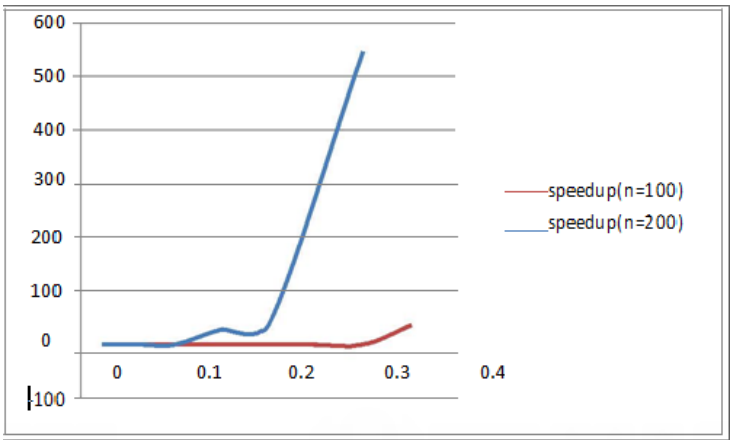


FIGURE 3- SPEED UP V/S DENSITY FOR *ALGO-2*

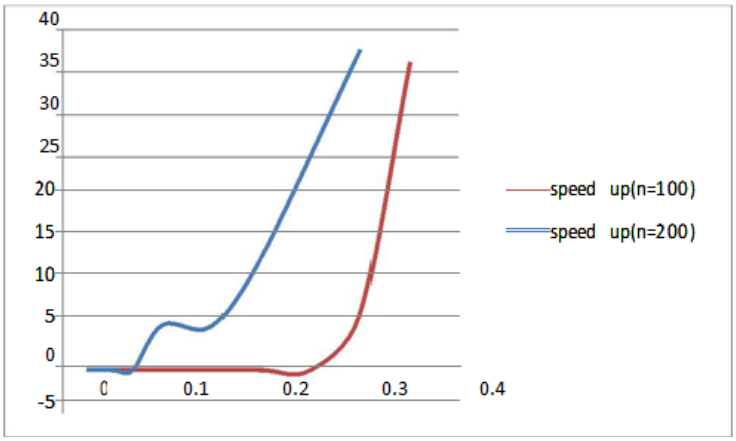


FIGURE 4-SPEED UP V/S DENSITY FOR *ALGO-3*

Maximum Clique problem can also be solved in parallel using a number of processors on a CREW-PRAM model. Some deterministic NC-algorithms exist for several important graph classes with a polynomially bounded number of maximal cliques (maximal independent sets). Moreover, two variants of a tabu search heuristics, a deterministic one and a probabilistic one can also be used to compute the maximum clique. We propose to investigate these aspects in future.

ACKNOWLEDGEMENT

I would like to extend my gratitude to all those people who helped and supported me in completing this paper. I would like to thank my professor, Dr.K.K.Shukla, for his lessons, guidance and advices. He inspired me to work efficiently on this project. In addition, I would like to thank him for motivating me to work hard in achieving my goals in life.

REFERENCES

[1] Bomze, I. M.; Budinich, M.; Pardalos, P. M.; Pelillo, M., "The maximum clique problem", *Handbook of Combinatorial Optimization* **4**, Kluwer Academic Publishers, pp. 1–74, 1999.

[2] Caprara and D. Pisinger and P . Toth "Exact Solutions on the Quadratic Knapsack Problem", *Informs Journal on Computing* , Vol. 11, No. 2, p.125 –137,1999.

[3] D.R. Wood, "An algorithm for finding a maximum clique in a graph", *Operations Research Letters*, Vol.21, p. 211 –217, 1997.

[4] DIMACS : *Dimacs Clique Benchmark Instances.* [online] Available: ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/

[5] E. Balas and C. S.Yu , "Finding a Maximum Clique in an Arbitrary Graph", *SIAM Journal Computing* , Vol.14, No.4, p.1054 – 1068, 1986.

[6] Gprof website: *Linux Tools Project - GProf Support*.[online] www. eclipse.org/li nuxtools/projectPages/gprof/

[7] Randy Carraghan and Panos M. Pardalos, "*An exact algorithm for the maximum clique problem*", Operations Research Letters, Volume 9 Issue 6, Pages 375-382, 1990.

