Parallel Search of Skewed Trees

T. Breitkreutz
T.A. Marsland
and
E. Altmann

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

Technical Report TR87.16

ABSTRACT

In our companion report on overheads in loosely coupled parallel systems, the need for a better sequential vertex covering algorithm, and for more complex graphs was demonstrated. Here we have explored schemes designed for the parallel search of skewed trees that arise from the use of an improved sequential algorithm. Three different biased binary multi-processor tree configurations are compared on a basis of time speedup, node count, and overheads in covering a representative set of computationally large graphs.

Acknowledgements

Financial support from the Canadian Natural Sciences and Engineering Research Council through Grant A7902 made the experimental work possible.

July 30, 1987

Parallel Search of Skewed Trees

T. Breitkreutz
T.A. Marsland
and
E. Altmann

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

Technical Report TR87.16

1. Introduction

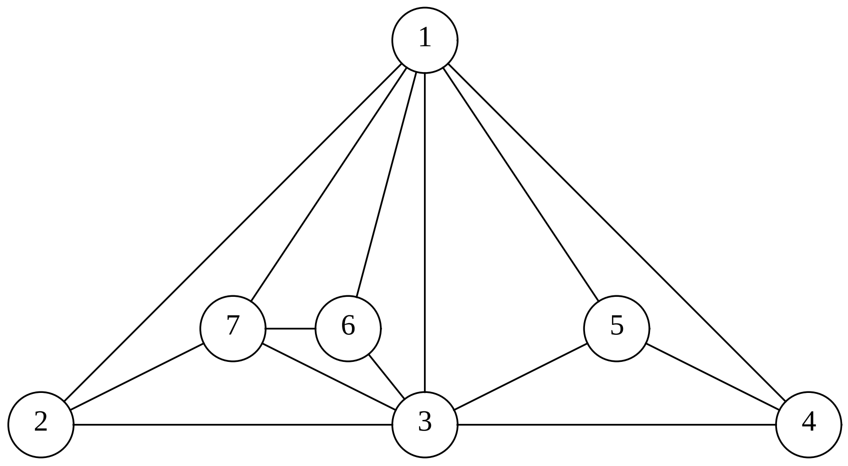
Highly skewed trees are particularly difficult to search in parallel. In branch-and-bound algorithms, skewing arises through the use of an improving bound which progressively cuts-off more of the tree as the search progresses. In game trees the bulk of the search is concentrated around the optimal variation, with significantly lower node density around the alternatives. In the simplest case of a minimal game tree, almost half of the search space is associated with the first variation [3]. Highly skewed trees arise in many branch-and-bound applications, but are particularly troublesome in computer chess, where real-time response is required. As a consequence of skewed trees, parallel searches either suffer from severe synchronisation and search overhead [4], or from the complexity of implementing schemes that might control these overheads [6]. This study examines a relatively simple scheme that attempts to match the available parallel architecture to the characteristics of the problem at hand, and examines the tradeoffs that occur between the principle overheads.

For this study, the vertex cover problem is used because it is a simple, well-defined problem that has biased search trees[1]. First, the straightforward, depth-one processor tree architecture of previous studies [1][8] is demonstrated to be ineffective on larger problems, because the vertex cover search tree is not uniform. Alternative forms of binary multi-processor tree architectures are presented and tested, and their speedups and overheads presented. Performance similar to a depth-one processor-tree searching uniform trees was demonstrated with the best configuration tested.

2. Vertex Cover Problem

The vertex cover problem is a classic from combinatorial graph theory. A subset of a graph is a vertex cover if it contains at least one vertex of every edge. An optimal vertex cover is one with the smallest possible number of vertices. A useful lower bound on the size of an optimal cover of a graph or subgraph is given by Wah and Ma [7] as the minimum number of vertices having a total outdegree greater than or equal to the number of edges in the graph. We are using a branch-and-bound algorithm similar to that given in Wah and Ma [7], using the above lower bound in three ways: a) to order the search in order of increasing lower bound, b) to cut-off unneccesary search when the lower bound exceeds the remaining depth in the search tree, and c) to detect solution, when the lower bound is zero.

Two search tree representations of the vertex cover algorithm are presented here. Figure 2.1 is a sample graph. The complete search tree in Figure 2.2 represents selections of vertices in potential solutions. Arrows indicate the vertices which have already been considered higher in the tree and thus form permutations of selections already considered. The square nodes indicate solutions. In our previous report, elmination of these redundant selections was called *duplicate path elimination* (*dpe*). With *dpe*, the search tree is extremely skewed to the left. The partial search tree in Figure 2.3 represents choices to select or ignore vertices, and thus is a binary tree. The binary tree representation is skewed to the left also (thirty nodes on the left half, eighteen on the right, when completely expanded), but not nearly as much as the selection tree representation.



Optimal cover: 1, 3, 4, 7

Figure 2.1: Sample Graph

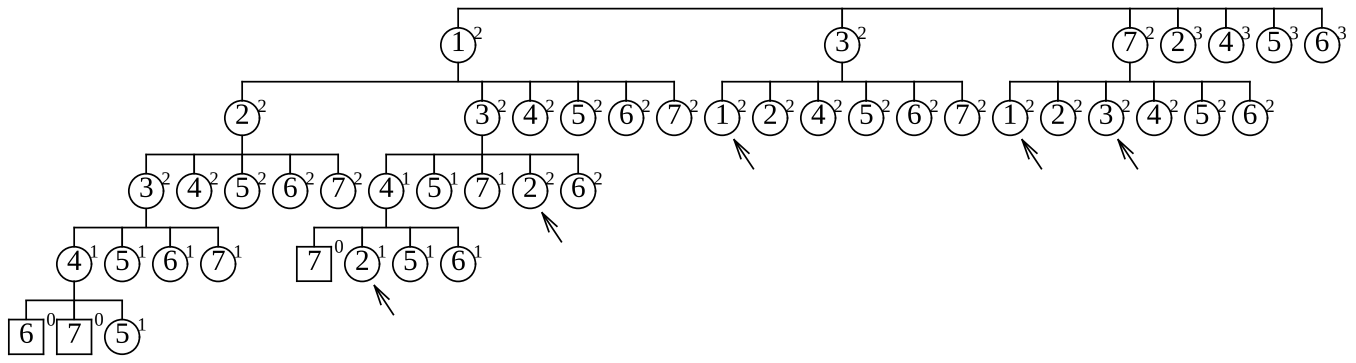


Figure 2.2: Selection Search Tree Representation

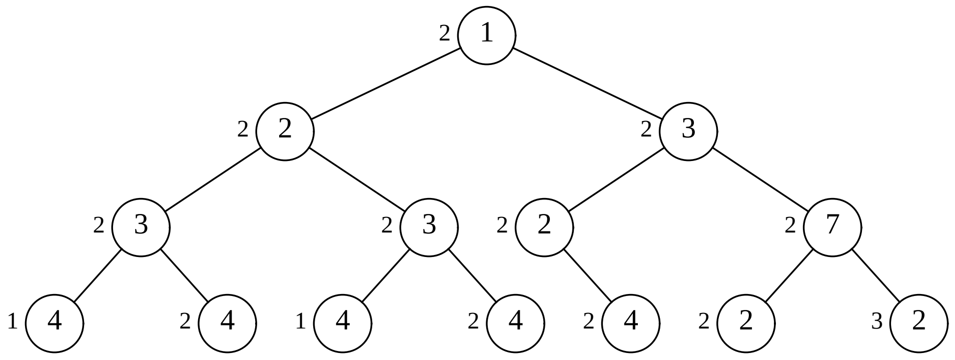


Figure 2.3: Binary Search Tree Representation

3. *Dynamic First Level* Multi-processor Architecture

In our previous report [1], and in Zariffa [8], the best multi-processor architecture for non-*dpe*, small graph problems was *Dynamic First Level* (DFL). As a preliminary study for this report, DFL was tested with *dpe* and with twelve of the computationally significant problems discussed in Section 5. The DFL architecture is inappropriate for such skewed trees because the subtrees from the first level are vastly different in size. Typically, the first processor will receive a piece of work much larger than the others. Regardless of the number of remaining processors, the first will be the last to finish. Table 3.1 contains the single processor times and the speedups for three, five, seven, and nine processor configurations. Please note that the times given in subsequent tables are smaller because the code was improved from the first implementation. The algorithm is identical.

	1 Proc.	Speedups			
Problem	(secs)	3	5	7	9
00	243	2.479	2.430	2.131	1.840
01	302	2.157	2.188	2.054	2.068
02	770	1.742	1.726	1.730	1.652
03	131	1.984	1.926	1.871	1.794
04	3608	1.517	1.515	1.514	1.513
05	3214	1.669	1.667	1.664	1.662
06	2214	1.564	1.563	1.561	1.558
07	4102	1.811	1.808	1.807	1.804
08	5484	1.266	1.264	1.263	1.261
09	5574	1.515	1.513	1.512	1.513
10	4945	2.451	2.415	2.098	1.658
11	2104	1.413	1.413	1.413	1.411
Mean	2724	1.797	1.786	1.718	1.645

Table 3.1: DFL Times and Speedups

Note the discrepancy between the speedups in Table 3.1 and those cited in Zariffa [8] (a speedup of 4.87 with seven processors). The difference aptly demonstrates how an inferior sequential algorithm can give the appearance of superior parallel performance, and that methods which work well on small problems may not be scalable upward.

4. Binary Biased Multi-processor Architecture

From Table 3.1 it is clear that the DFL architecture poorly matches the skew of the search tree. An effective processor tree architecture must overcome the great synchronisation costs evident with DFL. The binary biased processor tree architecture attempts to do this in two ways. First, it is designed around the less skewed binary representation of the search tree discussed in Section 2. Second, it allows the shape of the processor tree configuration to be matched to the application.

Figure 4.1 shows an example of a biased binary processor tree configuration with four processors. The nodes marked *1, 2,* and *1'* supervise the nodes marked *3, 2', 4,* and *1''*. Each number reprsents a single processor. In addition to doing actual work at the *1''* level, processor *1* is also supervising processors *2* and *4*. Processor *1* does not know about processor *3*, which is completely controlled by processor *2*.

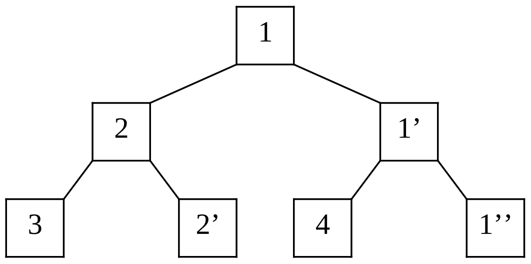


Figure 4.1: Shallow Binary Processor Tree (Configuration **A**)

Consider the processor tree in Figure 4.1 searching the search tree in Figure 2.3. The left processor subtree (*2*) first includes vertices 1, 2, 3, 4, etc., in it's attempted solution. The right processor subtree (*1'*) first excludes vertices 1, 3, 7, 2, etc. Both groups continue down the search tree until the global bound (the size of the currently best solution) is exceeded, until a new solution is found, or until the lower bounds exceed the remaining depth of the search tree (a lower bound cutoff). Then both groups will traverse the tree converging on one another somewhere in the middle. Within processor group *2*, the subtrees are split on the way down the search tree. That is, processor *3* will continue including all vertices, and processor *2'* will exclude all vertices after the first. When the processors reach the bottom of the search tree, they also converge together to complete the primary search subtree. On the other hand, with processor group *1'*, splitting is done on the way back up the search tree. That is, both processor *4* and *1''* exclude all vertices first, then backtrack up the search tree, splitting after including a vertex. In this way, the *2* processor group may start descending the right half of the search tree if it completes before the *1'* processor group.

As indicated above, there are two ways in which this architecture can match the search tree. The first is with supervisory and system overhead. In the processor tree given in Figure 4.1, Processors *3* and *4* are running at full speed, because they do not supervise any other processors. Processor *2'* is working at a speed of $1-\varepsilon$ where ε is the overhead involved in supervising one processor, and processor *1''* is operating at a speed of $1-2\varepsilon-\delta$ where δ is the operating system overhead. (See Section 6). In this way the supervisory and system overhead skews the processor tree slightly to the left. The other way the architecture can match the search tree is with the shape of the configuration. Figure 4.1 shows a *shallow* processor tree, as opposed to the *deep* processor tree shown in Figure 4.2. An alternate *deep* processor tree is shown in Figure 4.3. Obviously these three processor configurations will have different characteristics in searching trees: in Figure 4.2, the left half of the tree has an effective speed of $3-2\varepsilon$, and the right half has $1-\varepsilon-\delta$; in Figure 4.3, the left half has a speed of 1, and the right $3-3\varepsilon-\delta$.

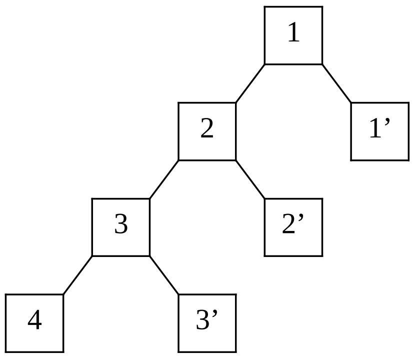


Figure 4.2: Deep Processor Tree (Configuration **C**)

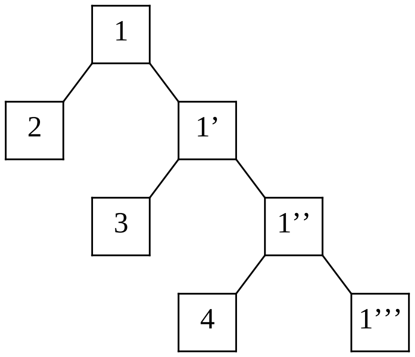


Figure 4.3: Alternate Deep Processor Tree

4.1. Experimental Configurations

This architecture, therefore, has many characteristics that should be considered in configuring it to the biased shape of the vertex cover search tree. We chose to examine processor trees with up to nine processors, each with the shallowest possible processor tree (**A**), the deepest processor tree (skewed left, **C**), and an intermediate configuration (**B**). The **C** configuration allows each processor to have only one assistant. In the **B** configuration, all processors have two assistants, except the first processor. If the number of processors in the configuration is odd, the deepest processor also has no assistant. There is no limitation of assistants in the **A** configuration, processors are included as shallow and as far left as possible. Sample configuration files for processor trees **A**, **B**, and **C** are given in Appendix A.

5. System Hardware and Software

For the experiments in this report, a set of eight standalone Motorola 68010 processors, and one 68010 running the UNIX† operating system were used. The *Virtual Tree Machine* [5] package was used for parallel startup, message passing, and limited system support on the standalones. Six of the the eight standalone processors were homogeneous. The other two were on older boards and were about 10% slower. The processor running UNIX was effectively even slower than the old standalones because of the operating system overhead. The single UNIX machine allowed for better monitoring of the system and control of experiments. It is placed as the primary processor so as to maintain the bias of the processor tree described in Section 4.

Section 3.2.2.4 of our companion study [1] examined the effect of including (doubling) the master process on the same machine as one assistant (slave) process. The inhomogeneity consequences decrease with the increasing use of processors. Furthermore, note that a good parallel algorithm should be able to overcome discrepancies in load caused by non-homogeneous processor systems. That is, an application that overcomes synchronisation overhead inherent in a problem should also compensate (within reason) for synchronisation overhead caused by the non-homogeneity of processors.

6. Problem Set

In the previous report [1] it was observed that the graphs being covered were too small to give reliable statistics on timing and overheads when *dpe* was used. For that reason a mechanism for developing problem sets which represent the whole range of possible graphs was developed. The problem set created consists of sixteen graphs each with 22 vertices. The sample problems are given in Appendix B, and optimal solutions in Appendix C.

† Registered trademark of AT&T in the USA and other countries.

6.1. Characteristics

There are two main characteristics of graphs which we chose to represent. The first is the density of connectivity, i.e. the number of edges compared to the number of vertices, or the mean outdegree. The second characteristic is the uniformity of the graph, i.e. the standard deviation of the outdegrees of the vertices. The sixteen graphs we used can be divided into four groups representing the range of densities, with the different levels of uniformity represented by the four graphs in each such group. Table 6.1 shows the characteristics of the graphs created for this experiment. The first four problems represent sparse graphs with 22 vertices, 46 edges, and a mean outdegree (average edges emanating from each node) of 4.18. The standard deviation of the outdegree varies from 1.14 to 2.59. Similarly for the other three groups which represent progressively denser graphs.

All graph problems have 22 vertices			
Problem	Number of Edges	Mean Outdegree	Standard Deviation
0	46	4.18	1.14
1			1.47
2			2.17
3			2.59
4	92	8.36	1.62
5			1.87
6			3.11
7			4.23
8	138	12.55	1.22
9			2.11
10			3.66
11			5.04
12	184	16.73	0.70
13			1.75
14			2.66
15			3.44

Table 6.1: Problem Set Statistics

6.2. Generation of Problems

The utility used to create the graphs was a simple program which first creates a minimally connected (very sparse) graph. The rest of the edges are added using the standard UNIX *random/srandom* random number generator, and two parameters to the program, each arbitrarily given the range of 0 to 100. The first parameter, the density, controls exactly the density of the produced graph. A density parameter of 0 will give the minimally connected graph made in the first stage of the program. A parameter of 50 will create a graph with an average outdegree of $\frac{n+1}{2}$, where n is the number of vertices. The second parameter controls the uniformity of the graph, and is a heuristic parameter (the standard deviation cannot be controlled exactly because of the random nature of the graph production process). For example, a uniformity parameter of 10 will create a nearly uniform graph, whereas a parameter of 90 will make a highly deviant graph. Therefore the increments of deviation in Table 6.1 are not constant, whereas those of the density are. The first four graphs had a density parameter of 20, the second four 40, the third four 60, and the last four 80. The first graph in each group had a uniformity parameter of 20, the second in each group 40, the third 60, and the fourth 80.

7. Experimental Results

The sixteen sample problems were solved by each configuration (**A, B,** and **C**) at parallelisms one to nine. The timing, speedups, node counts, and overheads are given in separate tables for configurations **A**, **B**, and **C**. Note that with one and two processors, only one configuration is possible. These are included in all three tables for comparison. The third column of the **A** and **B** tables are also identical, because there are only two possible configurations with three processors.

7.1. Timing (Total Overhead)

Tables 7.1a, 7.1b, and 7.1c contain the times for solutions using configurations **A**, **B**, and **C**, respectively. From these tables it can be seen that the sparse graphs are easier to solve (take less time) than dense ones. Another trend is that, with the exception of the sparsest four graphs, the more deviant graphs are easier to solve. In the case of the sparsest four graphs, the less deviant graphs are solved most quickly. Furthermore, it is interesting to note that configurations with even numbers of processors perform better than those with odd. Also, problem number eleven seems to be somewhat of an anomaly. A possible explanation for the smaller solution times is statistical good fortune. For each category of graph, a bell curve exists for the ease of solution possible for any given problem of that category. Problem eleven appears to be the most deviant of the problems within it’s own class’ relative ease of solution. One way to find out for certain would be to expand the problem set within each group, so that there are more problems representing each set of parameter values. Another would be to generate more graphs and to select for each category those that are average with respct to the ease of solution. Both of these endeavours would be time consuming.

	Number of Processors								
Problem	1	2	3	4	5	6	7	8	9
00	177	83	55	56	59	55	52	39	39
01	247	120	74	70	71	66	66	52	50
02	299	139	111	88	105	85	77	69	70
03	226	149	113	114	118	109	82	68	71
04	2511	1531	960	754	755	755	739	522	516
05	2102	1061	983	551	744	561	560	426	460
06	1479	842	671	456	480	454	531	409	358
07	1328	661	439	369	473	288	295	255	285
08	3150	1894	1214	1023	1022	964	1002	708	733
09	2900	1518	1331	773	822	793	795	793	809
10	2681	1314	1146	688	770	668	683	627	643
11	796	400	266	232	265	224	223	160	171
12	3094	1777	1286	995	995	935	976	722	733
13	3289	1878	1375	1056	1056	998	1039	777	808
14	3025	1654	1317	853	854	802	837	774	810
15	2635	1520	1069	828	835	778	814	604	629
Total	29939	16541	12410	8906	9424	8535	8771	7005	7185
Mean	1871	1033	775	556	589	533	548	437	449

Table 7.1a: Times for Configuration **A** (seconds)

Problem	Number of Processors								
	1	2	3	4	5	6	7	8	9
00	177	83	55	47	68	65	67	65	66
01	247	120	74	75	92	92	97	90	94
02	299	139	111	73	116	112	120	114	120
03	226	149	113	93	122	117	123	124	122
04	2511	1531	960	959	1471	1429	1450	1438	1444
05	2102	1061	983	747	1041	973	1038	979	1034
06	1479	842	671	674	830	774	914	773	827
07	1328	661	439	423	603	562	615	565	606
08	3150	1894	1214	1213	1867	1738	1863	1767	1780
09	2900	1518	1331	1336	1483	1381	1465	1381	1460
10	2681	1314	1146	1146	1208	1084	1148	1084	1149
11	796	400	266	266	355	332	354	334	354
12	3094	1777	1286	1282	1743	1633	1729	1629	1731
13	3289	1878	1375	1379	1840	1724	1823	1732	1825
14	3025	1654	1317	1322	1623	1515	1607	1516	1608
15	2635	1520	1069	1076	1494	1401	1489	1399	1485
Total	29939	16541	12410	12111	15956	14932	15902	14990	15705
Mean	1871	1033	775	756	997	933	993	936	981

Table 7.1b: Times for Configuration **B** (seconds)

Problem	Number of Processors								
	1	2	3	4	5	6	7	8	9
00	177	83	79	77	72	59	62	60	64
01	247	120	94	94	93	87	99	85	101
02	299	139	131	137	113	117	122	115	121
03	226	149	131	123	123	124	128	117	123
04	2511	1531	1466	1483	1472	1482	1469	1387	1471
05	2102	1061	1041	1050	1045	980	1043	982	1044
06	1479	842	832	835	983	780	837	786	834
07	1328	661	609	613	614	580	614	581	611
08	3150	1894	1864	1865	1866	1751	1866	1751	1866
09	2900	1518	1480	1494	1481	1396	1470	1396	1482
10	2681	1314	1199	1215	1208	1152	1215	1141	1207
11	796	400	358	358	357	338	359	334	361
12	3094	1777	1746	1748	1889	1648	1747	1647	1747
13	3289	1878	1843	1844	1844	1737	1839	1739	1845
14	3025	1654	1628	1625	1625	1534	1622	1534	1625
15	2635	1520	1492	1507	1499	1412	1499	1412	1498
Total	29939	16541	15993	16068	16284	15177	15991	15067	16000
Mean	1871	1033	999	1004	1017	948	999	941	1000

Table 7.1c: Times for Configuration **C** (seconds)

7.2. Node Counts (Search Overhead)

Tables 7.2a, 7.2b, and 7.2c contain node counts for the uniprocessor column, and search overhead factors for the multiprocessor columns, derived simply by dividing the multiprocessor node counts by the uniprocessor node counts. Node counts given are *expanded* node counts ([1], Section 3.2.2); that is, they are the nodes in the search tree actually expanded. The alternative, *generated* nodes, are those for which a lower bound is calculated, before pruning. As with the timing results, the node counts indicate that the most sparse group of four graphs was by far the easiest to solve; and show the preference for deviant graphs over uniform graphs except in the sparsest case.

	Nodes	Search Overhead							
Problem	1	2	3	4	5	6	7	8	9
00	3520	1.007	1.015	1.039	1.579	1.567	1.619	1.629	1.709
01	5767	1.005	1.009	1.018	1.656	1.515	1.547	1.555	1.700
02	7775	1.003	1.007	1.018	1.850	1.572	1.349	1.350	1.475
03	6111	1.452	1.763	2.126	2.948	3.222	2.544	2.261	2.413
04	115214	1.017	1.062	1.036	1.409	1.343	1.342	1.367	1.533
05	89795	1.000	1.000	1.001	1.624	1.449	1.450	1.288	1.569
06	57483	1.073	1.147	1.207	1.821	1.692	1.966	1.704	1.838
07	51550	1.000	1.002	1.002	1.653	1.373	1.375	1.218	1.418
08	160509	1.000	1.000	1.000	1.331	1.332	1.332	1.333	1.566
09	139273	1.000	1.000	1.001	1.426	1.424	1.425	1.425	1.717
10	126747	0.924	1.020	0.973	1.429	1.311	1.308	1.372	1.636
11	26045	1.001	1.002	1.005	1.559	1.565	1.567	1.302	1.386
12	158501	1.000	1.000	1.001	1.340	1.341	1.341	1.342	1.575
13	172273	1.000	1.000	1.000	1.338	1.338	1.339	1.339	1.581
14	150087	1.000	1.000	1.000	1.389	1.383	1.384	1.383	1.666
15	125420	1.000	1.000	1.001	1.458	1.302	1.302	1.302	1.612
Mean	87254	1.030	1.064	1.089	1.613	1.546	1.512	1.448	1.650

Table 7.2a: Search Overhead for Configuration **A**

	Nodes	Search Overhead							
Problem	1	2	3	4	5	6	7	8	9
00	3520	1.007	1.015	1.065	1.871	2.494	2.625	2.871	2.980
01	5767	1.005	1.009	1.036	1.775	2.069	2.282	2.384	2.474
02	7775	1.003	1.007	1.029	1.552	2.045	2.201	2.403	2.626
03	6111	1.452	1.763	1.692	2.667	2.904	3.311	3.312	3.444
04	115214	1.017	1.062	1.058	1.300	1.318	1.398	1.408	1.616
05	89795	1.000	1.000	1.002	1.446	1.360	1.628	1.669	1.784
06	57483	1.073	1.147	1.232	1.617	1.549	1.869	1.865	2.060
07	51550	1.000	1.002	1.003	1.373	1.319	1.480	1.692	1.788
08	160509	1.000	1.000	1.000	1.332	1.208	1.347	1.357	1.581
09	139273	1.000	1.000	1.000	1.424	1.240	1.440	1.446	1.682
10	126747	0.924	1.020	1.023	1.215	1.126	1.333	1.339	1.453
11	26045	1.001	1.002	1.008	1.563	1.488	1.682	1.708	1.754
12	158501	1.000	1.000	1.000	1.341	1.260	1.357	1.363	1.585
13	172273	1.000	1.000	1.000	1.338	1.263	1.348	1.446	1.552
14	150087	1.000	1.000	1.000	1.383	1.272	1.400	1.414	1.654
15	125420	1.000	1.000	1.001	1.301	1.266	1.477	1.483	1.620
Mean	87254	1.030	1.064	1.072	1.531	1.574	1.761	1.823	1.978

Table 7.2b: Search Overhead for Configuration **B**

	Nodes	Search Overhead							
Problem	1	2	3	4	5	6	7	8	9
00	3520	1.007	1.032	1.868	2.095	2.187	2.395	2.559	2.768
01	5767	1.005	1.015	1.910	2.270	2.343	2.305	2.570	2.617
02	7775	1.003	1.013	1.840	2.103	2.178	2.286	2.414	2.616
03	6111	1.452	1.663	2.536	2.810	2.879	3.057	3.212	3.421
04	115214	1.017	0.994	1.367	1.608	1.620	1.632	1.638	1.720
05	89795	1.000	1.001	1.624	1.754	1.766	1.797	1.839	1.901
06	57483	1.073	1.126	1.736	2.101	2.074	2.118	2.164	2.240
07	51550	1.000	1.001	1.652	2.134	2.152	2.212	2.299	2.373
08	160509	1.000	1.000	1.330	1.570	1.574	1.591	1.615	1.653
09	139273	1.000	1.000	1.425	1.714	1.724	1.746	1.781	1.833
10	126747	0.924	0.881	1.337	1.486	1.532	1.552	1.554	1.600
11	26045	1.001	1.003	1.826	2.174	2.278	2.225	2.292	2.397
12	158501	1.000	1.000	1.339	1.572	1.580	1.598	1.624	1.662
13	172273	1.000	1.000	1.337	1.579	1.595	1.598	1.630	1.667
14	150087	1.000	1.000	1.388	1.665	1.673	1.690	1.727	1.769
15	125420	1.000	1.000	1.458	1.610	1.676	1.701	1.741	1.785
Mean	87254	1.030	1.046	1.623	1.890	1.927	1.969	2.041	2.126

Table 7.2c: Search Overhead for Configuration **C**

From Tables 7.2a/b/c, it can be seen that the **A**, **B**, and **C** configurations have similar search overheads, except in the sparsest group of four graphs. One reason is the relative sizes of the processor trees and the search trees. Another is the fact that the easier graphs are be more susceptible to search overheads, because good bounds take longer to distribute over the multi-processor system compared to the over-all solution time. That is, for smaller problems, a proportionally larger time is spent searching before good bounds are propagated through the multiprocessor system.

7.3. Speedups

Tables 7.3a, 7.3b, and 7.3c contain the speedups calculated from the times given in Section 7.1 through simple division. The performance of configuration **A** is much better than **B** or **C**, and all are an improvement on the DFL results of Figure 3.1. The better performance is probably because **A** fits the search tree the best. Consider the binary representation of the search tree given in Figure 2.1, and the **A** configuration in Figure 4.1. The full binary representation of Figure 2.1 has thirty nodes on the left and eighteen on the right. Therefore, slightly more than half of the processing power is applied to the left half of the search tree (with thirty nodes), and slightly less than half of the processing power is applied to the right half (with eighteen nodes). At the second level of the tree, the splitting is similar. Furthermore, the largest (9 processor) **C** configuration is nearly as deep as the search trees, and thus decreases the effectiveness of the splitting. It might, however, find a near optimal solution more quickly than the other configurations.

Note that acceleration anomalies [2] occur with two and three processors, but not more. For configuration **C**, speedup is uniformly poor. Configuration **B** is at its best with three processors, but configuration **A** shows steady improvement in performance up to nine processors.

	Number of Processors								
Problem	1	2	3	4	5	6	7	8	9
00	1.000	2.132	3.218	3.160	3.000	3.218	3.403	4.538	4.538
01	1.000	2.058	3.337	3.528	3.478	3.742	3.742	4.750	4.940
02	1.000	2.151	2.693	3.397	2.847	3.517	3.883	4.333	4.271
03	1.000	1.516	2.000	1.982	1.915	2.073	2.756	3.323	3.183
04	1.000	1.640	2.615	3.330	3.325	3.325	3.397	4.810	4.866
05	1.000	1.981	2.138	3.814	2.825	3.746	3.753	4.934	4.569
06	1.000	1.756	2.204	3.243	3.081	3.257	2.785	3.616	4.131
07	1.000	2.009	3.025	3.598	2.807	4.611	4.501	5.207	4.659
08	1.000	1.663	2.594	3.079	3.082	3.267	3.143	4.449	4.297
09	1.000	1.910	2.178	3.751	3.527	3.656	3.647	3.656	3.584
10	1.000	2.040	2.339	3.896	3.481	4.013	3.925	4.275	4.169
11	1.000	1.990	2.992	3.431	3.003	3.553	3.569	4.975	4.654
12	1.000	1.741	2.405	3.109	3.109	3.309	3.170	4.285	4.221
13	1.000	1.751	2.392	3.114	3.114	3.295	3.165	4.232	4.070
14	1.000	1.828	2.296	3.546	3.542	3.771	3.614	3.908	3.734
15	1.000	1.733	2.464	3.182	3.155	3.386	3.237	4.362	4.189
Mean	1.000	1.869	2.556	3.322	3.081	3.484	3.481	4.353	4.255

Table 7.3a: Speedups for Configuration **A**

	Number of Processors								
Problem	1	2	3	4	5	6	7	8	9
00	1.000	2.132	3.218	3.765	2.602	2.723	2.641	2.723	2.681
01	1.000	2.058	3.337	3.293	2.684	2.684	2.546	2.744	2.627
02	1.000	2.151	2.693	4.095	2.577	2.669	2.491	2.622	2.491
03	1.000	1.516	2.000	2.430	1.852	1.931	1.837	1.822	1.852
04	1.000	1.640	2.615	2.618	1.707	1.757	1.731	1.746	1.738
05	1.000	1.981	2.138	2.813	2.019	2.160	2.025	2.147	2.032
06	1.000	1.756	2.204	2.194	1.781	1.910	1.618	1.913	1.788
07	1.000	2.009	3.025	3.139	2.202	2.362	2.159	2.350	2.191
08	1.000	1.663	2.594	2.596	1.687	1.812	1.690	1.782	1.769
09	1.000	1.910	2.178	2.170	1.955	2.099	1.979	2.099	1.986
10	1.000	2.040	2.339	2.339	2.219	2.473	2.335	2.473	2.333
11	1.000	1.990	2.992	2.992	2.242	2.397	2.248	2.383	2.248
12	1.000	1.741	2.405	2.413	1.775	1.894	1.789	1.899	1.787
13	1.000	1.751	2.392	2.385	1.787	1.907	1.804	1.898	1.802
14	1.000	1.828	2.296	2.288	1.863	1.996	1.882	1.995	1.881
15	1.000	1.733	2.464	2.448	1.763	1.880	1.769	1.883	1.774
Mean	1.000	1.869	2.556	2.749	2.045	2.166	2.034	2.155	2.061

Table 7.3b: Speedups for Configuration **B**

Problem	Number of Processors 1	2	3	4	5	6	7	8	9
00	1.000	2.132	2.240	2.298	2.458	3.000	2.854	2.950	2.765
01	1.000	2.058	2.627	2.627	2.655	2.839	2.494	2.905	2.445
02	1.000	2.151	2.282	2.182	2.646	2.555	2.450	2.600	2.471
03	1.000	1.516	1.725	1.837	1.837	1.822	1.765	1.931	1.837
04	1.000	1.640	1.712	1.693	1.705	1.694	1.709	1.810	1.707
05	1.000	1.981	2.019	2.001	2.011	2.144	2.015	2.140	2.013
06	1.000	1.756	1.777	1.771	1.504	1.896	1.767	1.881	1.773
07	1.000	2.009	2.180	2.166	2.162	2.289	2.162	2.285	2.173
08	1.000	1.663	1.689	1.689	1.688	1.798	1.688	1.798	1.688
09	1.000	1.910	1.959	1.941	1.958	2.077	1.972	2.077	1.956
10	1.000	2.040	2.236	2.206	2.219	2.327	2.206	2.349	2.221
11	1.000	1.990	2.223	2.223	2.229	2.355	2.217	2.383	2.204
12	1.000	1.741	1.772	1.770	1.637	1.877	1.771	1.878	1.771
13	1.000	1.751	1.784	1.783	1.783	1.893	1.788	1.891	1.782
14	1.000	1.828	1.858	1.861	1.861	1.971	1.864	1.971	1.861
15	1.000	1.733	1.766	1.748	1.757	1.866	1.757	1.866	1.759
Mean	1.000	1.869	1.991	1.987	2.007	2.150	2.030	2.170	2.027

Table 7.3c: Speedups for Configuration **C**

7.4. Synchronisation Overhead

When both speedup and search overhead are considered, performance is still not linear with respect to parallelism. The unaccounted for time lies in communication overhead and synchronisation overhead [1]. A multiplicative synchronisation/communication overhead is given in Tables 7.4a, 7.4b, and 7.4c, and is calculated with the following formula:

$$O_{synch} = \frac{T_P}{O_{search} \, \dfrac{T_1}{P}}$$

where O_{synch} is the synchronisation and communication overhead, T_P is the actual time taken to complete the problem by the processor tree, O_{search} is the search overhead, P is the degree of parallelism, and T_1 is the sequential (one processor) solution time. Note that some of the communication overhead is also disguised as search overhead, in that a pair of processors do the same work to reach the same point in the search tree, rather than communicate the complete state information when they split. In any case, earlier studies [4] have shown message passing communication overhead to be negligible.

From the synchronisation overheads it can again be seen that for configuration **A** the overheads are almost uniform among the different types of graph. Furthermore, for configurations **B** and **C**, the synchronisation overhead is a serious problem inhibiting the solution of non-sparse graphs, and yet it is not a factor with the sparsest set of four graphs. This probably arises because when the search tree is deeper and takes longer, the long idle times inherent in unbalanced (skewed too far) trees become apparent.

	Number of Processors								
Problem	1	2	3	4	5	6	7	8	9
00	1.000	0.931	0.918	1.218	1.056	1.190	1.270	1.082	1.160
01	1.000	0.967	0.891	1.114	0.868	1.058	1.209	1.083	1.072
02	1.000	0.927	1.106	1.156	0.949	1.085	1.336	1.368	1.428
03	1.000	0.908	0.851	0.949	0.886	0.898	0.998	1.065	1.172
04	1.000	1.199	1.080	1.159	1.067	1.343	1.535	1.217	1.206
05	1.000	1.010	1.403	1.047	1.090	1.105	1.286	1.259	1.255
06	1.000	1.061	1.187	1.022	0.891	1.089	1.278	1.298	1.185
07	1.000	0.995	0.990	1.109	1.077	0.948	1.131	1.261	1.362
08	1.000	1.203	1.156	1.299	1.219	1.379	1.672	1.349	1.337
09	1.000	1.047	1.377	1.065	0.994	1.152	1.347	1.535	1.462
10	1.000	1.061	1.257	1.055	1.005	1.140	1.363	1.364	1.319
11	1.000	1.004	1.001	1.160	1.068	1.079	1.251	1.235	1.395
12	1.000	1.149	1.247	1.285	1.200	1.352	1.647	1.391	1.354
13	1.000	1.142	1.254	1.284	1.200	1.361	1.651	1.411	1.398
14	1.000	1.094	1.306	1.128	1.016	1.150	1.399	1.480	1.447
15	1.000	1.154	1.217	1.256	1.087	1.361	1.661	1.408	1.333
Mean	1.000	1.053	1.140	1.144	1.042	1.168	1.377	1.300	1.305

Table 7.4a: Synchronization and Communication Overheads for Configuration **A**

	Number of Processors								
Problem	1	2	3	4	5	6	7	8	9
00	1.000	0.931	0.918	0.997	1.027	0.883	1.009	1.023	1.126
01	1.000	0.967	0.891	1.172	1.049	1.080	1.205	1.223	1.384
02	1.000	0.927	1.106	0.949	1.250	1.099	1.276	1.269	1.375
03	1.000	0.908	0.851	0.973	1.012	1.070	1.151	1.325	1.411
04	1.000	1.199	1.080	1.444	2.253	2.591	2.891	3.254	3.203
05	1.000	1.010	1.403	1.419	1.712	2.042	2.123	2.232	2.482
06	1.000	1.061	1.187	1.480	1.735	2.027	2.315	2.242	2.443
07	1.000	0.995	0.990	1.270	1.654	1.925	2.190	2.012	2.297
08	1.000	1.203	1.156	1.540	2.225	2.740	3.073	3.307	3.217
09	1.000	1.047	1.377	1.843	1.796	2.304	2.456	2.635	2.694
10	1.000	1.061	1.257	1.671	1.854	2.154	2.249	2.416	2.655
11	1.000	1.004	1.001	1.326	1.427	1.682	1.851	1.965	2.282
12	1.000	1.149	1.247	1.657	2.100	2.513	2.883	3.090	3.177
13	1.000	1.142	1.254	1.677	2.091	2.490	2.878	2.913	3.218
14	1.000	1.094	1.306	1.748	1.940	2.362	2.656	2.835	2.892
15	1.000	1.154	1.217	1.632	2.179	2.520	2.678	2.864	3.131
Mean	1.000	1.053	1.140	1.425	1.707	1.968	2.180	2.288	2.437

Table 7.4b: Synchronization and Communication Overheads for Configuration **B**

	Number of Processors								
Problem	1	2	3	4	5	6	7	8	9
00	1.000	0.931	1.297	0.932	0.971	0.914	1.024	1.060	1.176
01	1.000	0.967	1.125	0.797	0.829	0.902	1.217	1.071	1.406
02	1.000	0.927	1.298	0.996	0.899	1.078	1.249	1.275	1.392
03	1.000	0.908	1.046	0.858	0.968	1.143	1.297	1.289	1.432
04	1.000	1.199	1.762	1.728	1.823	2.186	2.509	2.698	3.065
05	1.000	1.010	1.484	1.230	1.417	1.584	1.933	2.032	2.351
06	1.000	1.061	1.499	1.301	1.582	1.526	1.870	1.965	2.266
07	1.000	0.995	1.374	1.118	1.083	1.218	1.463	1.522	1.745
08	1.000	1.203	1.775	1.781	1.887	2.119	2.606	2.754	3.225
09	1.000	1.047	1.531	1.446	1.490	1.675	2.032	2.162	2.509
10	1.000	1.061	1.523	1.356	1.516	1.683	2.044	2.191	2.532
11	1.000	1.004	1.345	0.985	1.031	1.118	1.419	1.465	1.703
12	1.000	1.149	1.693	1.688	1.942	2.023	2.473	2.622	3.058
13	1.000	1.142	1.681	1.677	1.775	1.987	2.449	2.595	3.029
14	1.000	1.094	1.615	1.548	1.613	1.819	2.221	2.349	2.733
15	1.000	1.154	1.699	1.569	1.767	1.918	2.341	2.462	2.866
Mean	1.000	1.053	1.484	1.313	1.412	1.556	1.884	1.969	2.280

Table 7.4c: Synchronization and Communication Overheads for Configuration **C**

8. Conclusion

Most methods of searching game trees are directional, introducing a bias towards initial branches, which in turn causes synchronisation overhead in parallel implementations. A simple scheme to match the bias of the search tree with a biased processor tree was attempted as part of a solution method for the vertex cover problem. Many interesting qualities of three different biased binary multi-processor tree configurations were demonstrated, including some trade-offs between search overhead and synchronisation overhead. On a wide variety of graphs, a shallow multi-processor tree showed speedups continuing to improve up to nine processors, and consistently similar search and synchronisation overheads. More distinctions between classes of problem graphs showed up in the less efficient configurations.

References

1. E. Altmann, T. Breitkreutz and T.A. Marsland, Overheads in Loosely Coupled Parallel Search, TR87.15, Comp. Sci. Dept., Univ. of Alberta, Edmonton, July 1987.
2. Ten-Hwang Lai and Sartaj Sahni, Anomalies of Parallel Branch-and-Bound Algorithms, *Communications of the ACM 27(6)*, (June 1984), 594-602.
3. T.A. Marsland and M. Campbell, Parallel Search of Strongly Ordered Game Trees, *Computing Surveys 14(4)*, (1982), 533-551.
4. T.A. Marsland, M. Olafsson and J. Schaeffer, Multiprocessor Tree-Searching Experiments, in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, 1985, 37-51.
5. M. Olafsson and T.A. Marsland, A Unix Based Virtual Tree Machine, *CIPS Congress 85*, Montreal, June 1985, 176-181.
6. Jonathan Schaeffer, Improved Parallel Alpha-Beta Search, *Proceedings Fall Joint Computer Conf.*, Dallas, Nov. 1986, 519-527.
7. Benjamin W. Wah and Y. W. Eva Ma, MANIP -- A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems, *IEEE Trans. on Computers C-33(5)*, (May 1984), 377-390.
8. N. Zariffa, Implementation and Analysis of Three Parallel Branch-and-Bound Algorithms for the Vertex Covering Problem, M.Sc. Thesis, School of Computer Science, McGill University, Montreal, March 1986.

Appendix A: Configuration Files

The following are the configuration files used for the experiments in this report. The files consist of processor names and their groupings. The processor named *sunshine* is a Sun 2/50 running Sun/UNIX 3.2, The rest are standalone processors: *vtm200* through *vtm203*, Sun-1s, and *vtm204* through *vtm209*, Sun-2s. Connection between two processors in the configuration files are indicated by listing them after each other. If a processor has more than one assistant, all except the last assistant must be delimited by braces. Within the braces is a separate subtree described in the same way. For example, the processor tree given in Figure 4.1 would be described as:

```
{
    1 {
        2
        3
    }
    4
}
```

indicating that processor `1` is assisted by processors `2` and `4`, and that processor `3` is an assistant to processor `2`.

1 Processor Configuration

```
{
    sunshine
}
```

2 Processor Configuration

```
{
    sunshine
    vtm204
}
```

3 Processor Configuration **A**

```
{
    sunshine
    vtm204
    vtm205
}
```

3 Processor Configuration **C**

```
{
    sunshine {
        vtm204
    }
    vtm205
}
```

4 Processor Configuration **A**

```
{
    sunshine {
        vtm204
        vtm205
    }
    vtm206
}
```

4 Processor Configuration **B**

```
{
    sunshine {
        vtm204
    }
    vtm205
    vtm206
}
```

4 Processor Configuration **C**

```
{
    sunshine
    vtm204
    vtm205
    vtm206
}
```

5 Processor Configuration **A**

```
{
    sunshine {
        vtm204 {
            vtm205
            vtm206
        }
    }
    vtm207
}
```

5 Processor Configuration **B**

```
{
    sunshine {
        vtm204 {
            vtm205
            vtm206
        }
        vtm207
    }
}
```

5 Processor Configuration **C**

```
{
    sunshine
    vtm204
    vtm205
    vtm206
    vtm207
}
```

6 Processor Configuration **A**

```
{
    sunshine {
        vtm204 {
            vtm205
            vtm206
        }
        vtm207
    }
    vtm208
}
```

6 Processor Configuration **B**

```
{
    sunshine {
        vtm204 {
            vtm205 {
                vtm206
            }
            vtm207
        }
        vtm208
    }
}
```

6 Processor Configuration **C**

```
{
    sunshine
    vtm204
    vtm205
    vtm206
    vtm207
    vtm208
}
```

7 Processor Configuration **A**

```
{
    sunshine {
        vtm204 {
            vtm205
            vtm206
        }
        vtm207
    }
    vtm208
    vtm209
}
```

8 Processor Configuration **A**

```
{
    sunshine {
        vtm204 {
            vtm205
            vtm206
        }
        vtm207
    } {
        vtm208
        vtm209
    }
    vtm200
}
```

9 Processor Configuration **A**

```
{
    sunshine {
        vtm204 {
            vtm205
            vtm206
            vtm207
        }
        vtm208
    } {
        vtm209
        vtm200
    }
    vtm201
}
```

Appendix B: Problem Set

The problems are presented here as hexadecimal forms of the square binary connection matrices. Each group of hexadecimal digits represents one row in the matrix. The sample graph from Figure 2.1 is given first as a connection matrix, then in hexadecimal.

Sample Graph (binary):

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

Sample Graph (hexadecimal):

```
3f 51 6f 54 58 51 72
```

Problem 00:

```
109080 202800 00104b 023822
045420 008088 212040 0200a4
14900c 2e2010 140510 020850
000013 000830 214000 088404
064101 001f01 092000 006040
0c0200 080230
```

Problem 01:

```
00442b 0006c8 030114 010200
080141 0c4000 000502 211004
000080 004163 000672 308880
140800 0a9000 10241a 121801
201812 0808a0 300080 084000
2098a0 221040
```

Problem 02:

```
02a948 002220 0040c0 001440
20000c 002404 202000 08150e
318046 044008 200088 054000
100108 204250 08080c 2c2103
100004 000102 225a84 0360aa
006054 000040
```

Problem 03:

```
088ac4 018188 20c209 008400
004000 108402 3d61da 0aa084
00c000 000003 200249 050200
280c04 108001 30c000 208808
00000c 008000 188860 204220
019000 081900
```

Problem 04:

```
0e6f08 00d434 2132dd 2011aa
203e8b 089689 1164af 308847
2a8a65 1f09d0 2272a0 338040
2b2900 241284 0f9910 087418
14a80a 1810c6 2f8065 18e11b
06c034 0be00c
```

Problem 05:

```
059831 08e6a8 12a7ac 20331f
090ad2 22b4e4 391508 100b0a
1d06ca 25848b 224482 19ba49
1e6486 0cc004 1b3a24 032410
390083 26004f 1cf410 0d0392
067a34 241430
```

Problem 06:

```
1a7206 28220e 37bfe6 0b4f01
2cb343 0c2602 0a3f32 242044
3bc103 2a811c 0c8654 0d8b01
3f8d49 0eb610 080003 0a4a22
08804f 009900 101224 385828
3ba0e1 0626a2
```

Problem 07:

```
063832 004477 044527 29c32c
200424 040425 044420 1c84af
20003c 200a24 201022 1bc0bd
0410fd 0c0024 00466c 1002ab
3fffdf 30262c 0466f4 1f77bb
384864 194664
```

Problem 08:

```
1bdf15 272b3a 25a9e4 1b9cb7
351dfc 3ee250 2d1a4f 212fe9
194d95 2686fa 3ee0af 2673eb
31d4e7 3a6433 0e7e7a 0bd69d
1e5f92 3731e5 12dcc4 2eaa5b
149fa5 24ef56
```

Problem 09:

```
0f2ddf 0f223f 3186bc 305b2f
30453e 3875aa 081a57 071355
311fec 05ef8e 24b715 2b3bfb
1cfdb2 277e3e 29367f 20e492
1f2799 3acfe3 3f35a2 3ef981
3797d9 34ccb6
```

Problem 10:

```
180060 2fd8d2 35fff7 1b25e7
1598f0 1eb77e 1b3b4f 181fef
0d84e1 1bc9d1 1ad7ce 0d69dc
09c8ea 0ddcf3 1e7f6d 3fffbf
2f63c4 1b1540 01cec4 0dcceb
1dcb45 0cf1c6
```

Problem 11:

```
1fffff 2e5ef8 371d72 3a547e
3dffff 2a1510 221028 36189a
221420 3fefff 3a55f0 3f39ff
321042 2b1cd3 325d48 3e1fbe
3ebc5e 3f5d6e 36d4f1 261472
2e5774 221508
```

Problem 12:

```
1eed7f 2fde3d 37dfca 3afbfb
3d78ff 1aff77 3d775f 3fbfcc
27c7ff 1fcff5 3f50ff 39f1f7
1df1ff 2df6fd 0e7f7b 2fffb6
373fcf 37bfce 3eebb6 33ff7a
2faefd 37bfa2
```

Problem 13:

```
07f7ed 0ff2f9 12a7dd 31dfff
39ff7f 36dd1e 3f773e 37bdbe
3add7e 37eefd 0777ef 2ffbf7
3e9cff 2fecde 3c5f4f 3e3f9f
36fe1f 1ff76f 3ffbf6 2ffffb
07effd 3e1ef6
```

Problem 14:

```
1d1edd 29371b 37ffff 2b4fee
0d7fbd 3eeeee 095fcb 0faf9d
1b4faf 3a8fbf 2ff3f9 3ff3ff
3ffdff 1efebf 2fff7f 2d8eab
0f3fdf 3a5faf 3ffff7 2f77bb
1db7fc 3afffc
```

Problem 15:

```
1efff7 2dfe97 37ffff 3afbb7
2dc7ae 1a8e96 3f7fff 3ebfbe
3cc7e6 3ccad6 3dd6ff 3bebfe
3ffdff 2ee6fe 3fff7f 28bfb6
2eefc6 3ddfc6 0acf87 3ffffb
3ffffd 3c8a8e
```

Appendix C: Solutions

Listed here are the solutions for the problem set found by the sequential algorithm:

Problem 00:

```
0 2 3 4 5 7 8 10 12 15 16 17
```

Problem 01:

```
0 1 2 3 4 6 7 9 10 14 15 16
```

Problem 02:

```
0 1 3 7 8 11 13 14 15 17 18 19
```

Problem 03:

```
0 1 6 7 10 11 12 16 18 20 21
```

Problem 04:

```
0 2 3 4 5 6 7 8 9 11 12 14 16 17 18 19
```

Problem 05:

```
0 1 2 3 4 5 6 7 8 11 14 17 18 19 20 21
```

Problem 06:

```
0 2 3 4 6 8 9 11 12 15 17 18 19 20 21
```

Problem 07:

```
0 1 2 3 7 8 10 11 12 15 16 18 19 21
```

Problem 08:

```
0 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 21
```

Problem 09:

```
0 1 2 3 6 7 8 9 10 11 13 14 16 17 18 19 20
```

Problem 10:

```
1 2 3 5 6 7 9 10 11 14 15 16 17 18 20 21
```

Problem 11:

```
0 1 2 3 4 7 9 10 11 13 15 16 17 18 20
```

Problem 12:

```
0 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 18 19 20
```

Problem 13:

```
0 1 2 3 4 6 7 9 10 11 12 13 15 16 17 18 19 20 21
```

Problem 14:

```
0 1 2 4 5 7 9 10 11 12 13 14 15 16 18 19 20 21
```

Problem 15:

```
0 1 2 3 4 6 7 8 10 11 12 13 14 15 17 18 19 20
```

Appendix D: Utilities

In the process of working with vertex cover graphs and examining their characteristics, a graphic tool was developed to edit, manipulate, and possibly regularise graphs, written in the SunView 3.2 system.

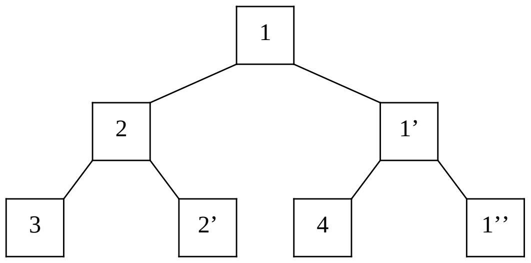


Figure for Appendix A.

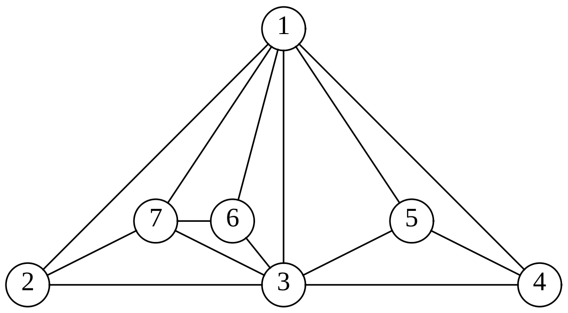


Figure for Appendix B