Electronic Preprint for Journal of Information Processing Vol.30

Regular Paper

FPGA Acceleration of Swap-Based Tabu Search for Solving Maximum Clique Problems

Kenji Kanazawa[1,a)]

Received: December 15, 2021, Accepted: March 16, 2022

Abstract: The Swap-Based Tabu Search (SBTS) is a heuristic algorithm for solving the maximum independent set problems and can solve the maximum clique problems as well because the maximum clique in a graph is equivalent to the maximum independent set in its complementary graph. Although SBTS is a powerful algorithm in solving the maximum clique problems and has abundant inherent parallelism, it is difficult to parallelize because of its solution searching heuristic involving indirect indexing on array components. In this paper, we show a variant of SBTS that does not require indirect indexing while maintaining the same accuracy as that of the original version of SBTS and describe its hardware acceleration using a Field-Programmable Gate Array (FPGA). Experimental results show that our proposed SBTS variant on FPGA can solve the maximum clique problems up to 51.1 times faster than the original SBTS algorithm on CPU and up to 5.40 times faster than our proposed SBTS variant on CPU, respectively.

Keywords: FPGA, maximum clique problems, tabu search

1. Introduction

Given an undirected graph G, the maximum clique problem aims to find a clique with the maximum possible number of vertices for G where a clique is G's subgraph in which every two distinct vertices are adjacent (joined by an edge). The maximum clique problem is an NP-hard combinatorial optimization problem that occurs in many practical applications [1] such as bioinformatics [2] and VLSI CADs [3]. The maximum independent set problem involves finding an independent set with the maximum possible size in a graph, where the independent set comprises pairwise non-adjacent vertices in the graph. In general, finding a clique in G is equivalent to finding an independent set in G's complementary graph. Herein, G's complementary graph $\overline{G}$ indicates a graph obtained by disjointing all the adjacent vertices and jointing every two vertices that originally have not been adjacent in G. Therefore, a maximum clique in G can be obtained by converting G to $\overline{G}$ and then finding a maximum independent set in $\overline{G}$. **Figure 1** displays an example of the maximum clique in G and the maximum independent set in $\overline{G}$. In Fig. 1, the black vertices consist of the maximum clique in G and the maximum independent set in $\overline{G}$, respectively. As depicted in Fig. 1, the maximum independent set in $\overline{G}$ consists of the same vertices that are comprised of the maximum clique in G.

The Swap-Based Tabu Search (SBTS) [4] is one of the best performing heuristic algorithm for solving the maximum clique problems as well as the maximum independent set problems [4], [5]. SBTS has abundance of inherent parallelism. However, it involves many occurrences of irregular access to array components

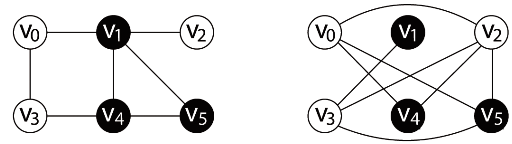


Fig. 1 G and its maximum clique (on the left side), and $\overline{G}$ and its maximum independent set (on the right side).

caused by indirect indexing in the form of $X[Y[i]]$ in its solution search heuristic, thereby making its parallel implementation difficult.

In this paper, we describe a modified version of the SBTS algorithm that does not require indirect indexing while maintaining the same algorithmic accuracy as the original SBTS algorithm and also represent its parallel implementation on a Field Programmable Gate Array (FPGA). We then evaluate the performance of our modified algorithm and its FPGA implementation, evaluate the effectiveness of our proposed approach, based on which we discuss the future tasks. This paper is an extension of our previous conference papers [6], [7]. In Refs. [6], [7], we proposed two techniques for facilitating parallel processing of the SBTS algorithm, showed an FPGA implementation of the modified SBTS based on the two techniques, and evaluated overall performance gain by the FPGA implementation. In this paper, we evaluate the individual performance gain by our proposed techniques on software and FPGA, respectively, based on which we discuss how much does each technique contributes to the overall performance gain.

2. Related Work

Several accelerators for the maximum clique problems have been proposed up to now.

A GPU implementation of a parallel algorithm called the

[1] Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] kanazawa@cs.tsukuba.ac.jp

© 2022 Information Processing Society of Japan

"sticker model" was proposed [8], and its performance on Nvidia GTX-680 was compared to several exact algorithms on software [9], [10] using the DIMACS benchmark graphs [11]. This algorithm was capable of solving problem instances up to 10× faster than [9] and up to 35× faster than [10], keeping the same solution accuracy as obtained by those exact approaches. However, this approach is considerably inferior to the latest heuristic algorithms on software [4], [5] in terms of both execution time and solution accuracy.

In our earlier research, we proposed an FPGA solver for the maximum clique problems encoded into the partial maximum satisfiability problems (Partial MaxSAT) based on the Dist algorithm [12], which is a local search algorithm for solving Partial MaxSAT [13]. The experimental results obtained using the DIMACS benchmark graphs showed that the solver has ability to solve the maximum clique problems encoded into Partial MaxSAT instances up to 22× faster than the same algorithm on software. However, this approach is not yet comparable to the latest heuristic algorithms including SBTS in terms of solution accuracy.

Neither the above-mentioned accelerators are comparable to the latest heuristic algorithms on software in terms of execution time or solution accuracy. Our proposed approach on FPGA achieves up to 51.1 times speedup without deteriorating the solution accuracy compared with SBTS.

3. The Swap-Based Tabu Search Algorithm

This section introduces the Swap-based tabu search (SBTS) proposed by Jin and Hao [4].

3.1 Definitions of the Symbols

Algorithm 1 summarizes the main procedure of SBTS. In Algorithm 1, V and E are the sets of vertices and edges in a graph G, respectively. S denotes an independent set in G, S_{max} the largest independent set found so far, $|S|$ the size of S, i.e., the number of vertices in S, and $|S_{max}|$ the size of S_{max}. NS_k ($k = 0, 1, 2, > 2$) displays the subset of the difference set of V and S (denoted by $V\backslash S$), in which the element vertices have k adjacent vertices in S. Note that $NS_{>2}$ represents a subset of $V\backslash S$, in which the element vertices have three or more adjacent vertices in S. **Figure 2** shows an example of the independent set. In Fig. 2, the sets of vertices, i.e., S and NS_k, are as follows: $S = \{v_2, v_3, v_6\}$, $NS_0 = \{v_0\}$, $NS_1 = \{v_5, v_7, v_8\}$, $NS_2 = \{v_4\}$, and $NS_{>2} = \{v_1\}$.

SBTS searches for a solution by iteratively moving a vertex in any of NS_k to S and its adjacent vertices in S to $V\backslash S$, i.e., "swapping" a vertex in any of NS_k and its adjacent vertices in S. Here, k is called the *mapping degree* and $m(v)$ denotes the mapping degree of a vertex v in $V\backslash S$. $m(v) - 1$ is equal to the decrease in $|S|$ if v in $V\backslash S$ is moved to S because v in $N\backslash S$ and its adjacent $m(v)$ vertices in S are swapped. Note that $|S|$ is increased only if v is selected from NS_0, i.e., the set of vertices whose mapping degrees are equal to 0. Thus, the mapping degree indicates the change in solution accuracy when a swapping is executed. $e(v)$, the *expanding degree* of a vertex v in S, represents the number of $v \in S$'s adjacent vertices v' in $V\backslash S$ such that $m(v') = 1$. For $v \in S$, $e(v) - 1$ corresponds to the number of vertices whose mapping degrees become zero if v is moved to $V\backslash S$ by swapping. Therefore, selecting a vertex in $V\backslash S$ whose adjacent vertex in S has a larger expanding degree as a candidate for swapping leads to reach a larger independent set in the next iteration. $d(v)$, the *diversifying degree* of a vertex v in $V\backslash S$, indicates the number of v's adjacent vertices in $V\backslash S$. **Table 1** displays the mapping, expanding, and diversifying degrees of the vertices in Fig. 2.

$t(v)$ indicates the iteration number until which v in $V\backslash S$ is prohibited from moving back to S. For example, $t(v) = 100$ represents that v is prohibited from moving back to S during $Iter \leq 100$, where $Iter$ represents the current iteration number. We call $t(v)$ v's "tabu tenure", and if $Iter > t(v)$, we say that v has passed its tabu tenure.

Algorithm 1 Main procedure of the SBTS algorithm

```
Require: A graph G = (V, E) and Iters_max
    /* V and E: sets of vertices and edges in G, respectively. */
    /* Iters_max: maximum iterations per run). */
Ensure: The largest independent set S_max found.
 1: /* Considering an initial independent set. */
 2: Init(S, NS_0, NS_1, NS_2, NS_>2, m(), d(), e(), t());
 3: S_max ← S; |S_max| = |S|;
 4:
 5: /* Searching a Solution. */
 6: for Iter = 1 to Iters_max do
 7:     v̇ ← NULL;
 8:     if NS_0 ≠ ∅ or NS_1 ≠ ∅ then
 9:         /* Intensification */
10:         v̇ ← Sel_Intense(S, NS_0, NS_1, e(), d(), t(), Iter);
11:         if v̇ == NULL then
12:             /* Diversification */
13:             v̇ ← Sel_Diversify(S, NS_2, NS_>2, d(), t(), Iter);
14:         end if
15:     end if
16:     updateIndependentSet(v̇, S, NS_0, NS_1, NS_2, NS_>2);
17:     updateParameters(m(), d(), e(), t());
18:     if |S| > |S_max| then
19:         S_max ← S; |S_max| ← |S|;
20:     end if
21: end for
22: return S_max;
```

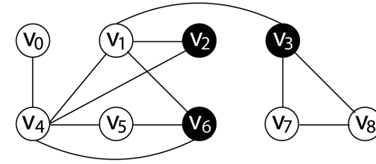


Fig. 2 Example of the independent set (indicated by black vertices).

Table 1 Mapping, expanding, and diversifying degrees of the vertices in Fig. 2.

	mapping	diversifying
v_0	0	1
v_1	3	1
v_4	2	3
v_5	1	1
v_7	1	1
v_8	1	1

	expanding
v_2	0
v_3	2
v_6	1

3.2 Overall Procedure

3.2.1 Considering an Initial Independent Set

SBTS begins by considering an initial independent set. First, it sets S to empty. Then, until V becomes empty, it iteratively

© 2022 Information Processing Society of Japan

Table 2 New mapping and diversifying degrees of w and changes in the mapping and diversifying degrees of w', and w''.

	w	w'	w''
mapping	1	+1	−1
diversifying	$L(w) - 1$	−1	+1

selects $v \in V$ at random, moves v to S, and removes all of the v's adjacent vertices from V. The algorithm then restores V to its original state, calculates $V \backslash S$, and then divides the element vertices in $V \backslash S$ into NS_0, NS_1, NS_2, and $NS_{>2}$ according to their mapping degrees.

3.2.2 Searching a Solution

Subsequently, SBTS iteratively alternates the *intensification phase* (searching for a better solution than the current one) and *diversification phase* (perturbing the current solution to escape from local optima) until it determines an independent set with the target size or reaches the iteration limit.

In both of the search phases, the search process is driven by selecting a vertex in any of NS_k and then moving it to S. Hereinafter, $\dot{v}$ denotes a vertex in any of NS_k that is selected to move to S at the current iteration. As mentioned in Section 3.1, when $\dot{v}$ is moved to S, its k adjacent vertices in S must be moved to $V \backslash S$ at the same time, except that $\dot{v}$ is selected from NS_0. $|S|$ is increased, i.e., the solution is improved, only if $\dot{v}$ is selected from NS_0. If $\dot{v}$ is selected from NS_1, then the search moves to another solution without deteriorating $|S|$. Otherwise, $|S|$ is decreased.

After moving $\dot{v}$ to S, the diversifying and mapping degrees of the following vertices as well as $\dot{v}$ are changed.

- $\dot{v}$'s adjacent vertices that have moved to $V \backslash S$ (denoted by w).
- $\dot{v}$'s adjacent vertices that have originally stayed in $V \backslash S$ (denoted by w').
- Vertices in $V \backslash S$ that are adjacent to w (denoted by w'').

These vertices are moved to any of NS_k ($k = 0, 1, 2, > 2$) in accordance with their new mapping degrees. **Table 2** shows new mapping and diversifying degrees of w and changes in the mapping and diversifying degrees of w', and w''. Note that values for w' and w'' represent the differences from their previous values and $L(w)$ represents the number of adjacent vertices of $\dot{v}$ in Table 2. $m(w)$ and $d(w)$ become 1 and $L(w) - 1$, respectively, because $\dot{v}$ has become the only adjacent vertex of w in S and all the other adjacent vertices of w have been in $V \backslash S$. $m(w')$ and $d(w')$ are incremented and decremented, respectively, because $\dot{v}$, i.e., one of their adjacent vertices that was originally in $N \backslash S$, has been moved to S. On the contrary, $m(w'')$ and $d(w'')$ are decremented and incremented, respectively, because w, i.e., their adjacent vertices originally in S, have been moved to $N \backslash S$.

The tabu tenures are calculated for w as follows:

- When $m(\dot{v}) = 1$:
  If $|NS_1| < |NS_2| + |NS_{>2}|$, then $t(w) = Iter + 10 + \mathcal{R}(|NS_1|)$, where $\mathcal{R}(x)$ is a random integer ranging from 0 to $x - 1$. Otherwise, $t(w) = Iter + |NS_1|$.
- When $m(\dot{v}) > 1$: $t(w) = Iter + 7$.

3.3 Heuristics for Selecting the Vertex to be Swapped

3.3.1 Intensification Phase

In the intensification phase, SBTS aims to find better solutions or to reach other solutions without deteriorating the current solution. For these purposes, $\dot{v}$ is selected from any of the vertices in either NS_0 or NS_1.

In the intensification phase, $\dot{v}$ is selected as follows.

(1) If NS_0 is not empty, then $\dot{v}$ is always randomly selected from NS_0 regardless of whether it has passed its tabu tenure.
(2) Otherwise, $\dot{v}$ is selected from NS_1 as follows.
  i. If $|NS_1| > |NS_2|+|NS_{>2}|$, the vertices in NS_1 whose adjacent vertex in S (denoted by u) satisfies $e(u) = 1$ are excluded from the candidates for selecting in advance.
  ii. Among the vertices in NS_1 that have passed their tabu tenures, select the vertex whose adjacent vertex in S has the largest expanding degree. If there are multiple candidate vertices whose adjacent vertex in S has the same expanding degree, then select the vertex that has the largest diversifying degree (ties are broken at random).

If there are no vertices that satisfy the aforementioned conditions, this implies that the search has reached a local optimum. At this point, SBTS switches the search to the diversification phase.

3.3.2 Diversification Phase

In the diversification phase, SBTS attempts to escape from local optima by perturbing the current solution. In the diversification phase, $\dot{v}$ is selected as follows.

(1) If $|NS_1| > |NS_2| + |NS_{>2}|$, then select $\dot{v}$ from the vertices in $NS_{>2}$ that have passed their tabu tenures with the largest diversifying degree (ties are broken at random).
(2) Otherwise,
  a. with probability p, select $\dot{v}$ from the vertices in NS_2 that have passed the tabu tenures with the largest diversifying degree (ties are broken at random).
  b. with probability $1 - p$, select $\dot{v}$ from $NS_{>2}$ at random without considering the tabu tenures.

3.4 Performance Bottleneck

To clear the performance bottleneck, we conduct performance profiling of the original SBTS algorithm published by the developers [*1] using DIMACS graph suite [11] on Core-i7 5820K 3.3 GHz with 32 GB main memory. In the performance profiling, we use 17 graphs for which the original SBTS requires more than 1 second to reach the best-known solutions. Herein, the best-known solution stands for a clique of the graph with the best-known size. For each graph, we execute 10 trials, measure their average time, and then analyze the breakdown of the average time. Each trial is stopped when it either finds the best-known solutions or reaches 10^9 iterations that are divided into 10^5 restarts (restart per 10^4 iterations). Trials that do not obtain the best-known solutions are excluded.

Table 3 represents the outlines of the graphs used for the profiling and the results of the trials. N_v and N_e denote the number of vertices and edges in each graph, #adj$_{avg}$ shows the average number of adjacent vertices of each vertex. "best" denotes sizes of the best-known solutions, i.e., the number of vertices in the best-known solutions. "size" denotes the best sizes of the obtained cliques in the experiments (the average sizes are in brack-

*1 http://www.info.univ-angers.fr/pub/hao/mis.html

© 2022 Information Processing Society of Japan

Table 3 Benchmark graphs and results of searching solutions by the original SBTS.

graph	N_v	N_e	#adj$_{avg}$	best	size	sec$_{avg}$
brock400_1	400	59,723	100.4	27	27 (27)	42.5
brock400_2	400	59,786	100.1	29	29 (29)	9.97
brock400_3	400	59,681	100.6	31	31 (31)	1.16
san400_0.7_1	400	55,860	119.7	40	40 (40)	2.03
brock800_1	800	207,505	280.2	23	23 (22.8)	3,597.0
brock800_2	800	208,166	278.6	24	24 (24)	1,065.3
brock800_3	800	207,333	280.7	25	25 (25)	1,100.4
brock800_4	800	207,643	279.9	26	26 (26)	263.0
C1000.9	1,000	450,079	98.8	68	68 (68)	6.89
san1000	1,000	250,500	498.0	15	15 (15)	23.0
MANN_a45	1,035	533,115	3.8	345	345 (345)	11.9
p_hat1500-1	1,500	284,923	1,119.1	12	12 (12)	8.65
C2000.5	2,000	999,836	999.2	16	16 (16)	2.87
MANN_a81	3,321	5,506,380	3.9	1,100	1,100 (1,100)	18.3
keller6	3,361	4,619,898	610.9	59	59 (59)	314.3
C4000.5	4,000	4,000,268	1,998.9	18	18 (18)	1,053.8
C2000.9	2,000	1,799,532	199.5	80	78 (77.2)	-

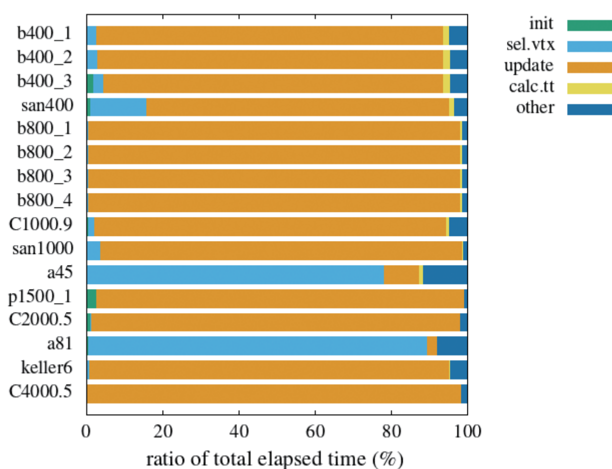


Fig. 3 Breakdown of total elapsed time for each graph in Table 3 (graph names are abbreviated).

ets). sec$_{avg}$ denotes the elapsed time in seconds to obtain the best-known solutions.

Figure 3 shows the breakdown of the total elapsed time for each graph. C2000.9 is excluded because no trials reach its best-known solution in the profiling. In Fig. 3, "init" denotes initialization that includes time spent for reading a given graph and converting to its complementary graph. "sel.vtx" denotes the time to select a vertex for swapping in the intensification and diversification phases, which is proportional to N_v. "update" represents the summation of the time to swap $\dot{v}$ and its adjacent vertices and the time to update the parameters (the mapping, expanding, and diversifying degrees) occurring with the swapping, which depends on #adj$_{avg}$ because the number of the parameters to be updated in each iteration is proportional to #adj$_{avg}$. "calc.tt" shows the time to calculate new values of the tabu tenures. Note that "update" does not include "calc.tt" in the profiling. "other" represents the subtractions of above-stated times from the total elapsed time.

As shown in Fig. 3, "update" occupies 82% on average and 99% at maximum over the total elapsed time, which forms the bottleneck in most cases. As for MANN graphs, "sel.vtx" dominates the total elapsed times, which occupies 78% and 89% over the total elapsed times of MANN_a45 and MANN_a81, respectively. This is because N_v is relatively large and #adj$_{avg}$ is considerably smaller than N_v in MANN graphs, thereby resulting in large ratio of "sel.vtx" over the total elapsed times.

4. Algorithm Modification for Parallel Processing

In this section, we explain how the algorithm is modified for facilitating the parallel processing. Based on our experiments, there is hardly any loss in solution accuracy by the alternations, depending on problem instances. This will be discussed in Section 6.

4.1 Change the Heuristic in the Intensification Phase

The most difficult part to parallelize is Step 2) in the intensification phase. As described in Section 3.3.1, when $\dot{v}$ is selected from NS_1, it is necessary to determine its adjacent vertex in S for each candidate vertex in NS_1 and then read its expanding degree. In software programs, a table of the expanding degrees and a list of adjacent vertices (called *adjacency list*) for each vertex are prepared so that the expanding degrees of only the adjacent vertices can be read immediately. This requires indirect array indexing in the form of $X[Y[i]]$, where X and Y correspond to the expanding degree table and a candidate vertex v's adjacency list, respectively, and $Y[i]$ represents one of v's adjacent vertices in S.

There are several issues that hinder parallelization of referring the expanding degree table. Firstly, the aforementioned indirect array indexing incurs irregular access patterns to $X[]$. Secondly, there can be multiple candidate vertices in NS_1 that have a common adjacent vertex in S, which causes access conflicts in the expanding degree table. It is difficult to schedule the vertices in NS_1 to avoid the access conflicts in advance because the adjacent vertex in S for each vertex varies in every iteration. We may avoid the access conflict by preparing duplicates of the expanding degree table. However, this requires a very complicated control logic for the parallel circuits to maintain the consistency between the duplicates.

To solve these issues, we introduce a substitute decision methodology without involving indirect indexing so that the vertex selection method can be parallelized more easily. Specifically, we utilize the number of iterations after passing the tabu tenures instead of the expanding degrees. The new vertex selection method in the intensification phase is as follows:

(i) If NS_0 is not empty, $\dot{v}$ is always randomly selected from NS_0 regardless of whether it has passed its tabu tenure.

(ii) Otherwise, it is selected from the vertices in NS_1 **that have taken the most iterations after passing their tabu tenures (ties are broken at random).**

Step (ii) can be executed by comparing the tabu tenures of the vertices in NS_1, selecting the vertex with the minimum tabu tenure, and comparing the selected vertex's tabu tenure with current *Iter* to check whether the selected vertex has passed its tabu tenure. This process does not incur the indirect array indexing because the tabu tenures are referred to directly by the candidate vertices.

4.2 Simplify the Tabu Tenure Calculation

To calculate the tabu tenures, it is necessary to calculate $\mathcal{R}(x)$, which is equivalent to the remainder of a random integer divided by x. The simple ways to calculate remainder are the restoring

© 2022 Information Processing Society of Japan

and non-restoring divisions. However, these consume calculation time proportional to the bit width of the operands.

In our proposed approach, we replace $\mathcal{R}(x)$ with a function that returns 0 or $x-1$ at random (denoted by $\mathcal{R}'(x)$). $\mathcal{R}'(x)$ can be realized as follows:

- If a random bit r is equal to 1, then return $x-1$. Otherwise, return 0.

$\mathcal{R}'(x)$ can be implemented only by a random bit generator and a multiplexer, which requires considerably less hardware resources and can be executed in a fixed time regardless of the operand bit width.

5. Hardware Implementation

5.1 Overview of the Hardware

Figure 4 shows a block diagram of the hardware of our proposed approach on an FPGA board. In Fig. 4, m_tbl, d_tbl, and tabu_tbl denote the mapping and diversifying degrees, and the tabu tenures for each vertex, respectively. S_array and NS_k_arrays denote the binary arrays which correspond to S and NS_k in Algorithm 1, respectively. |S|_cnt and $|\text{NS}_k|$_cnts denote the counters that hold the numbers of the elements in S and each of NS_k, respectively. The list table holds the adjacency list for each vertex. "list(v)" represents v's adjacency list. The address table translates v to the address of list(v) in the list table. "w_buf" is a temporal buffer for storing certain vertices. All the tables and buffers except for the list table are implemented by the on-chip memories on FPGA. "vertex selector" denotes the function block to select $\dot{v}$ based on the selection methods in the intensification or diversification phases. "updater" denotes the function block to update m_tbl, d_tbl, tabu_tbl, S_array, NS_k_arrays, |S|_cnt, and $|\text{NS}_k|$_cnts in each iteration.

The aforementioned tables and arrays, with the exception of the list and address tables and w_buf, are in the form of a content addressable memory (CAM), in which a vertex number is used as the address. For example, m_tbl[0] represents the mapping degree of Vertex 0, and tabu_tbl[1] indicates the tabu tenure of Vertex 1. The values in the arrays indicate which of the vertex sets each vertex is in. For example, S_array[2] = 1 indicates that Vertex 2 is in S, and NS_0_array[3] = 0 represents that Vertex 3 is not in NS_0. Furthermore, when S_array[v] and NS_k_array[v] become 1 and 0, respectively, this means that v has moved from NS_k to S at the iteration. By using such data structures, moving of the vertices among the vertex sets can be implemented using bit operations, thereby simplifying the circuit and reducing the on-chip memory consumption.

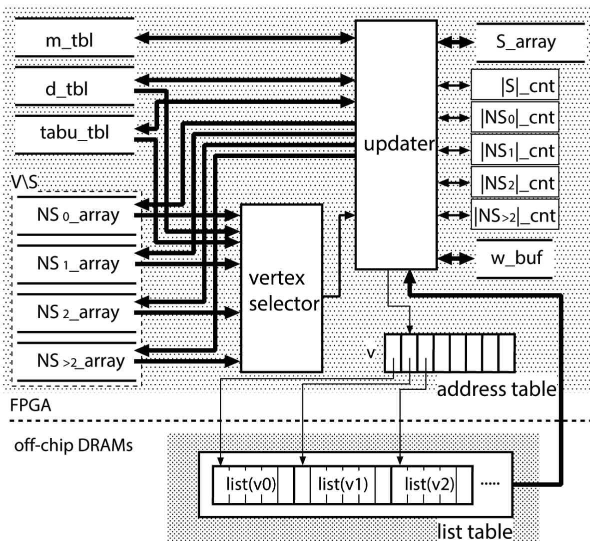


Fig. 4 Block diagram of the proposed approach on an FPGA board.

5.1.1 Parallel Processing of Tables and Arrays

m_tbl, d_tbl, tabu_tbl, S_array, NS_k_arrays, and w_buf are divided into P sub-banks such that up to P vertices in list(v) are processed at once. Also, |S|_cnt and $|\text{NS}_k|$_cnts comprise P sub-counters, respectively. The i-th sub-counters hold the number of 1 in the i-th sub-banks (i.e., the number of the element vertices in the i-th sub-banks) of the corresponding arrays. The total number of the element vertices is determined by summing up the values of the sub-counters.

To read out the values of those tables and arrays for a vertex, the lower $\log_2 P$ bits of the vertex number are used to specify the sub-banks to access and the remaining bits of the vertex are used as the address of the sub-banks. For example, m_tbl[v] is read out from the address $v \gg \log_2 P$ of the i-th sub-bank of m_tbl, where '$\gg$' represents the right shift and i is equivalent to v's lower $\log_2 P$ bits. For any vertices v in i-th sub-bank, the following equation holds: $v = i + P \times (v \gg \log_2 P)$.

list(v) whose length $L(v)$ exceeds P is divided into multiple sub-lists and each sub-list is processed in turn. Vertices in each sub-list are placed to the fixed position so that these can access only the corresponding sub-banks (i.e., the sub-banks whose indices are equivalent to the lower $\log_2 P$ bits of the vertices), thereby facilitating parallel processing of multiple vertices in adjacency lists.

5.2 Processing Sequence

In our proposed approach, a host CPU converts the given graph to its complementary graph, generates the address and list tables, and finally, downloads them to the circuit on FPGA.

Next, the circuit on FPGA constructs an initial solution as follows.

i) Set all the bits in NS_0_array to 1 and those in the other arrays to 0.
ii) Select $\dot{v}$ that satisfies NS_0_array[$\dot{v}$] = 1 at random and set NS_0_array[$\dot{v}$] and S_array[$\dot{v}$] to 0 and 1, respectively.
iii) Read every w' from list($\dot{v}$), and then update m_tbl[w'] and d_tbl[w'] and set NS_k_array[w'] according to the new values of m_tbl[w'].
iv) Update |S|_cnt and $|\text{NS}_k|$_cnts (i.e., increase or decrease the sub-counters of them according to the changes in S_array and NS_k_arrays by Steps ii) and iii)).
v) Repeat Steps ii) to iv) until all the bits in NS_0_array become 0.

Subsequently, the following procedure is executed on FPGA to search for a solution.

vi) Select $\dot{v}$ according to the heuristics in SBTS.
vii) Set S_array[$\dot{v}$] to 1 and NS_k_array[$\dot{v}$] to 0, respectively.
viii) Read every w and w' from list($\dot{v}$), and then update m_tbl[w],

© 2022 Information Processing Society of Japan

d_tbl[w], m_tbl[w'], and d_tbl[w']. Then, set S_array[w] to 0, and NS$_k$_array[w] and NS$_k$_array[w'] according to the new values of m_tbl[w] and m_tbl[w']. Concurrently, store every w into w_buf.

ix) Update |S|_cnt and |NS$_1$|_cnt.

x) For each vertex in w_buf, execute the following processing in turn (updating $m(w'')$, $d(w'')$, and NS$_k$_array[w'']).

(a) Read list(w). Herein, $\ddot{v}$ denotes the vertices in list(w).

(b) Update the values of m_tbl[$\ddot{v}$] and d_tbl[$\ddot{v}$].

(c) Set NS$_k$_array[$\ddot{v}$] according to the new values of m_tbl[$\ddot{v}$].

(d) Update S_cnt and |NS$_k$|_cnts.

xi) Update tabu_tbl[w].

5.3 Selection of a Vertex to be Swapped

In our proposed approach, the selection methods of $\dot{v}$ can be categorized into the following three types (note that the intensification phase is changed as described in Section 4.1):

x) Select a vertex among the vertices v that satisfy NS$_k$_array[v] = 1 at random, where k = 0 or >2 (corresponding to Step (1) in the intensification phase and Step (2)-b in the diversification phase).

y) Select a vertex with the smallest value of tabu_tbl among the vertices v that satisfy NS$_1$_array[v] = 1 (corresponding to Step (2) in the intensification phase).

z) Select a vertex with the largest value of d_tbl among the vertices v that satisfy NS$_k$_array[v] = 1, where k = 2 or >2 (corresponding to Steps (1) and (2)-a in the diversification phase).

Among them, (x) is parallelized using a binary tree of multiplexers with random selections signals. (y) is virtually equivalent to the min function, which is parallelized by a binary tree of the min operation circuits (comprising multiplexers and comparators). Similarly, (z) is parallelized by a binary tree of the max operation circuits.

In the following sections, we describe the details of the above-mentioned circuits. All of the circuits are fully pipelined.

5.3.1 Random Vertex Selector

Figure 5 shows the circuit for the random selection (P = 8). The area surrounded by the dotted line represents the sub-banks of NS$_k$_array (k = 0 or > 2) and their indices.

The random vertex selector selects a vertex by executing the following procedure.

(i) The values in the sub-banks of NS$_k$_array are read from the address pointed by the addr_pointer. Concurrently, the "MADs" (multiply-add operators) restore their associated vertices from the pairs of the read address and the sub-bank indices by calculating the following equation: $v = i + P \times addr_pointer$, where i stands for the sub-bank index. If the values read from the NS$_k$_array are 0, the vertices are invalidated (converted to NULL) by the "masks."

(ii) The vertices are sent to the multiplexer tree. Each multiplexer in the tree selects any of the valid (non-NULL) input vertices at random. The multiplexer at the final stage stores the input vertex to "reg." If there is already a vertex in the reg, it randomly selects either the vertex from its preceding multiplexer or one that has been stored in the reg and then overwrites the reg by the selected vertex.

(i) and (ii) are repeated N_v/P times by changing the addr_pointer from 0 to $N_v/P - 1$, a vertex is in this way randomly selected among the vertices in NS_k.

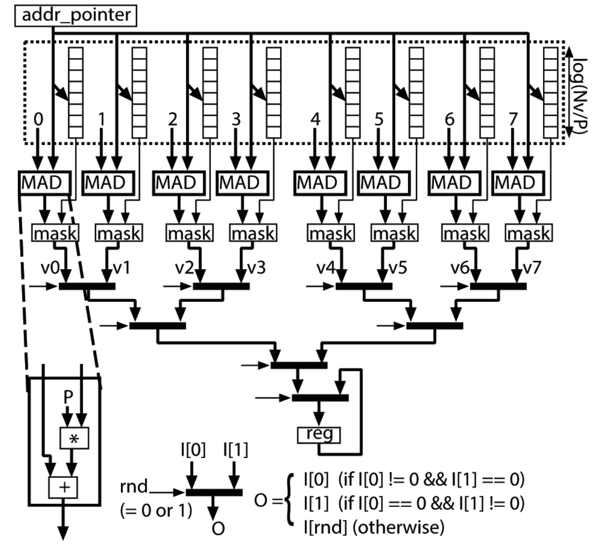


Fig. 5 Random vertex selector (P = 8).

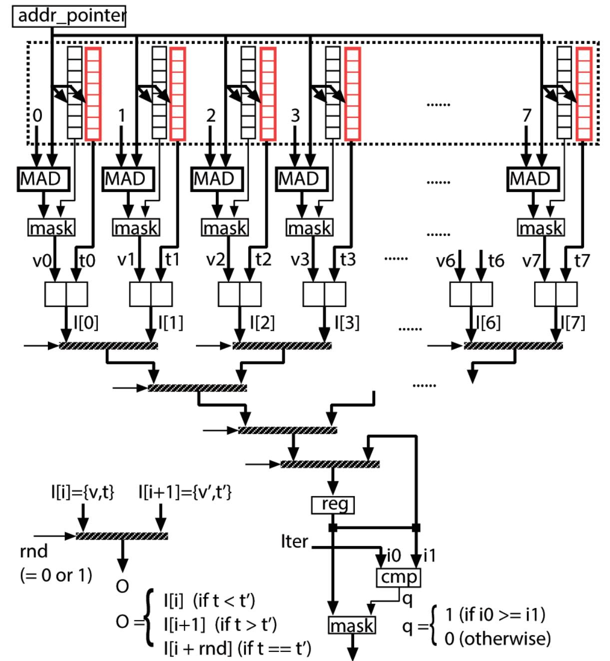


Fig. 6 Vertex selector based on the min function (P = 8).

5.3.2 Vertex Selector Based on the Min Function

Figure 6 shows the vertex selector based on the min function. In Fig. 6, the area surrounded by the dotted line indicates the sub-banks of NS$_1$_array (the black rectangles), the sub-banks of tabu_tbl (the red rectangles), and the indices of the sub-banks.

The procedure for the vertex selection is as follows.

(i) The values in the sub-banks of NS$_1$_array are read from the address pointed by the addr_pointer. Then, the corresponding vertices are extracted in the same manner as the random vertex selector. In addition, the tabu tenures of the extracted vertices are read from the tabu_tbl.

© 2022 Information Processing Society of Japan

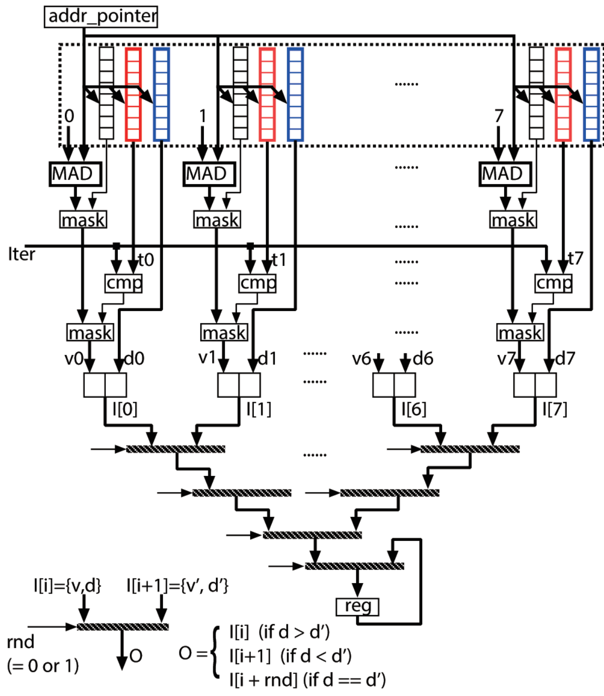


Fig. 7 Vertex selector based on the max function ($P = 8$).

(ii) The extracted vertices are sent to the multiplexer tree along with their tabu tenures. Each multiplexer in the tree selects the valid input vertex with smaller tabu tenure (ties are broken at random). The multiplexer at the final stage selects the vertex with the smaller tabu tenure among the valid vertex from its preceding multiplexer and the one that has been stored already in the reg, and then overwrites the reg by the selected one.

(i) and (ii) are repeated N_v/P times by changing the addr_pointer from 0 to $N_v/P-1$, the vertex with the smallest tabu tenure is in this way selected among the vertices in NS_1.

(iii) Finally, "cmp" compares the tabu tenure of the selected vertex to *Iter* to determine whether the selected vertex has passed its tabu tenure and discards it if it has not passed its tabu tenure.

5.3.3 Vertex Selector Based on the Max Function

Figure 7 shows the vertex selector based on the max function. The blue rectangles surrounded by the dotted line denote the sub-banks of d_tbl, the black rectangles are the sub-banks of NS_k_array (k = 2 or > 2), and the red rectangles are the same as in Fig. 6. The integer numbers are the indices of these sub-banks.

The procedure for the vertex selection is as follows. Firstly, (i) the values in the sub-banks of NS_k_array are read from the address pointed by the addr_pointer. Then the corresponding vertices are extracted in the same manner as the former two vertex selectors. In addition, (ii) the cmps compare the tabu tenures to *Iter* and then the masks at the lower positions invalidate the vertices that have not passed their tabu tenures. Then, (iii) the multiplexer tree selects the vertex with the largest diversifying degree among the valid vertices and the vertex that has been stored already in the reg, and then overwrites the reg by the selected one. (i)–(iii) are repeated N_v/P times by changing the addr_pointer from 0 to N_v/P, and in this way the vertex with the maximum diversifying degree is selected.

5.4 Parallel Processing of Vertex Swapping

5.4.1 Structure of the Updater

As mentioned in Section 3.2.2, the following values must be updated whenever $\dot{v}$ and w are swapped: the mapping and diversifying degrees of w, w', and w'', and the tabu tenures of w. Accordingly, it is necessary to update the values of m_tbl, d_tbl, tabu_tbl, S_array, and NS_k_arrays for the above vertices.

Figure 8 depicts the structure of the updater in Fig. 4, the circuit for updating m_tbl, d_tbl, tabu_tbl, S_array, NS_k_arrays, |S|_cnt, and $|\mathrm{NS}_k|$_cnts. In Fig. 8, tick-lined rectangles ("update_S", "update_NS_k", "+/-", "sum", and "calc_tt") represent the components of the updater. The blue rectangles denote the sub-counters of |S|_cnt and $|\mathrm{NS}_k|$_cnts. "ptr" indicates the number of the entries in each sub-buffer of w_buf. Also, $v_0, v_1, \ldots, v_P$ represent the vertices read from the list table in parallel.

"update_S" and "update_NS_k" are the functions for updating the values of S_array and the values of NS_k_arrays, respectively. "+/−" denotes a functional unit to calculate the new mapping and diversifying degrees and to update the values of the sub-counters. "sum" represents an adder tree to sum up the values of the sub-counters. "calc_tt" calculates the tabu tenures by executing $\mathcal{R}'(x)$ described in Section 4.2.

5.4.2 Procedure for Updating the Tables and Arrays

(1) Set m_tbl[$\dot{v}$] and d_tbl[$\dot{v}$] to 0 and $L(\dot{v})$, respectively. Herein, we suppose that $\dot{v}$'s mapping degree was λ before swapping and we denote $\dot{v}$'s lower $\log_2 P$ bits as α. Note that m_tbl[$\dot{v}$] is at address $\dot{v} \gg \log_2 P$ of α-th sub-bank in m_tbl as mentioned in Section 5.1.1, and the same is true for the other tables and arrays.

(2) Set S_array[$\dot{v}$] and NS_λ_array[$\dot{v}$] to 1 and 0, respectively. Also, increment α-th sub-counter of |S|_cnt and decrement α-th sub-counter of $|\mathrm{NS}_\lambda|$_cnt.

(3) Read the address table by $\dot{v}$ to read out the address of list($\dot{v}$), and then read list($\dot{v}$) from the list table.

(4) Update the tables and arrays for the vertices in list($\dot{v}$). As mentioned in Section 5.1.1, up to P vertices are read from list($\dot{v}$) at once and then following steps are executed for the vertices in parallel, which are repeated until all the vertices in list($\dot{v}$) are processed. Herein, we denote the vertices read from list($\dot{v}$) as $\ddot{v}$ and $\ddot{v}$'s lower $\log_2 P$ bits as β.

  (a) If S_array[$\ddot{v}$] = 1, then store $\ddot{v}$ into β-th sub-buffers of w_buf, increment their corresponding ptrs. Otherwise, execute the following steps. Note that $\ddot{v}$ with S_array[$\ddot{v}$] = 1 correspond to w and $\ddot{v}$ with S_array[$\ddot{v}$] = 0 correspond to w'.

    (i) Increment m_tbl[$\ddot{v}$] and decrement d_tbl[$\ddot{v}$]. Herein, we represent the new values of m_tbl[$\ddot{v}$] as μ.

    (ii) Update NS_k_arrays[$\ddot{v}$] and $|\mathrm{NS}_k|$_cnts as follows:

      (A) If μ = 1 or 2, then set NS_μ_array[$\ddot{v}$] to 1, and set $\mathrm{NS}_{\mu-1}$_array[$\ddot{v}$] to 0. Also, increment and decrement the corresponding sub-counters of $|\mathrm{NS}_\mu|$_cnt and $|\mathrm{NS}_{\mu-1}|$_cnt, respectively.

      (B) If μ = 3, then set $\mathrm{NS}_{>2}$_array[$\ddot{v}$] to 1, and set NS_2_array[$\ddot{v}$] to 0. Also, increment and decrement the corresponding sub-counters of $|\mathrm{NS}_{>2}|$_cnt and $|\mathrm{NS}_2|$_cnt, respectively.

© 2022 Information Processing Society of Japan

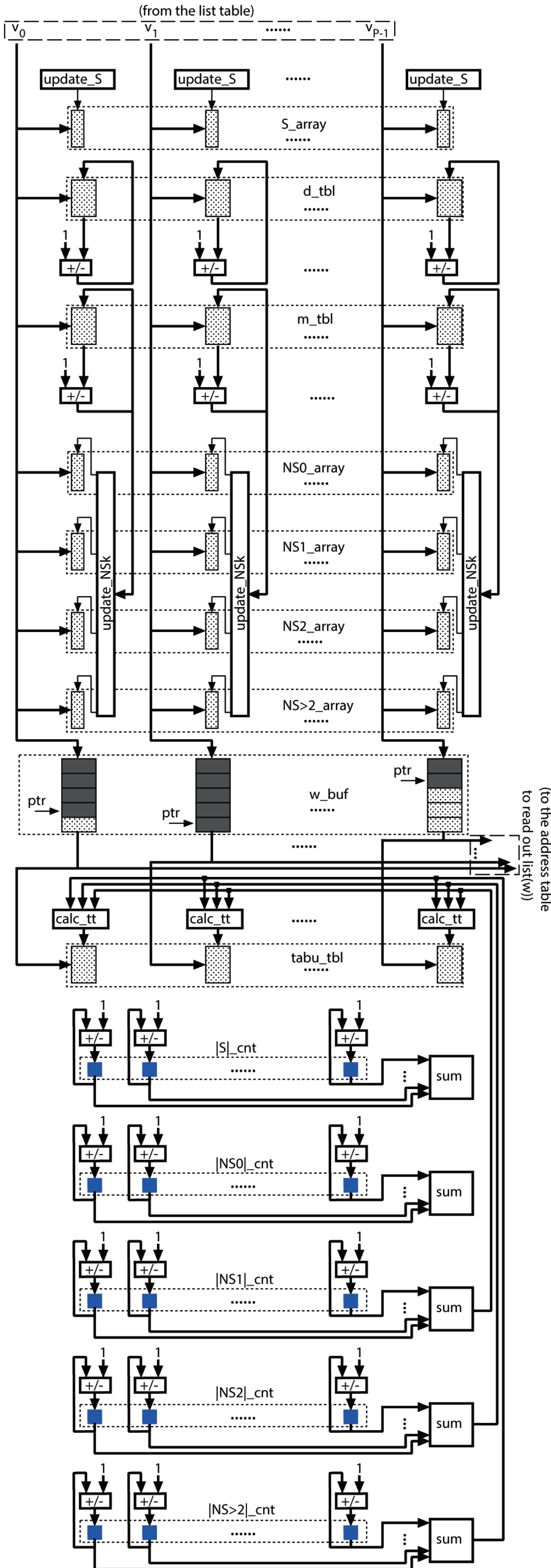


Fig. 8 Block diagram of the updater.

Table 4 Pipeline specifications.

	number of stages	initiation interval
(4)	7	1
(5)	7	1
(6)	5	3

(5) Execute the following steps for every vertex in w_buf:
  (a) Read a vertex w from a sub-buffer of w_buf.
  (b) Read list(w) in the same manner as Step 3). Note that the vertices in list(w) correspond to w''.
  (c) Update m_tbl, d_tbl, NS$_k$_arrays, and NS$_k$_cnts for w''. Likewise Step (4), up to P vertices are read from list(w) at once and then following steps are executed for the vertices in parallel, which are repeated until all the vertices in list(w) are processed.
    (i) Decrement m_tbl[w''] and increment d_tbl[w'']. Herein, we denote the new values of m_tbl[w''] as ν.
    (ii) Update NS$_k$_arrays[w''] and the sub-counters of |NS$_k$|_cnts as follows:
      (A) If $\nu = 0$ or 1, then set NS$_\nu$_array[$\ddot{v}$] to 1, and set NS$_{\nu+1}$_array[$\ddot{v}$] to 0. Also, increment and decrement the corresponding sub-counters of |NS$_\nu$|_cnt and |NS$_{\nu+1}$|_cnt, respectively.
      (B) If $\nu = 2$, then set NS$_2$_array[$\ddot{v}$] to 1, and set NS$_{>2}$_array[$\ddot{v}$] to 0. Also, increment and decrement the corresponding sub-counters of |NS$_2$|_cnt and |NS$_{>2}$|_cnt, respectively.

(6) Execute the following steps repeatedly until all the ptrs become 0:
  (a) Read out vertices in all the sub-buffers of w_buf whose corresponding ptrs > 0.
  (b) Calculate the tabu tenures for the read vertices w, write the calculated values into tabu_tbl[w], set S_array[w] to 0, and decrement ptrs if ptrs > 0.
  (c) Set NS$_1$_array[w] and NS$_0$_array[w] to 1 and 0, respectively. Also, increment and decrement the corresponding sub-counters of |NS$_1$|_cnt and |NS$_0$|_cnt, respectively.
  (d) Set m_tbl[w] and d_tbl[w] to 1 and L(w)−1, respectively.

(7) For |S|_cnt and |NS$_k$|_cnts ($k = 0, 1, 2, > 2$), sum up values of the sub-counters, respectively. All the summations are executed simultaneously.

Steps (4), (5), and (6) are pipelined, respectively. **Table 4** shows the specifications of the pipeline processing. In our current implementation, the pipeline for Step (6) stalls for three clock cycles to avoid control hazards while Steps (4) and (5) are fully pipelined.

6. Performance Evaluation

6.1 Experimental Setup

We compared our modified SBTS algorithm and its FPGA implementation with the original SBTS algorithm using DIMACS benchmark graph suite [11]. In order to clarify the contribution of each technique described in Sections 4.1 and 4.2, we prepared three variants of our modified SBTS algorithm on software as follows.

© 2022 Information Processing Society of Japan

Table 5 Resource utilization of FPGA-mod-A (90 MHz, $P = 64$).

Resource	Consumed / Total
LUT	346.6 K / 660.8 K (52.4%)
flipflop	319.1 K / 1,321 K (24.1%)
36 Kb block RAM	737 / 2,160 (34.1%)
DSP	12 / 5,520 (0.217%)

Table 6 Resource utilization of FPGA-mod-AB (140 MHz, $P = 64$).

Resource	Consumed / Total
LUT	257.3 K / 660.8 K (38.9%)
flipflop	256.9 K / 1,321 K (19.4%)
36 Kb block RAM	727 / 2,160 (33.7%)
DSP	12 / 5,520 (0.217%)

Table 7 The number of graphs for which each implementation found the best-known solutions in all the trials.

# of graphs	original	mod-A	mod-B	mod-AB	FPGA-mod-AB
80	78	77	77	77	76

- mod-A: SBTS with the modified heuristic for vertex swapping (described in Section 4.1);
- mod-B: SBTS with the simplified tabu tenure calculation (described in Section 4.2);
- mod-AB: SBTS with both of the modified heuristic for vertex swapping and the simplified tabu tenure calculation.

Also, we tested FPGA implementations based on mod-A and mod-AB, respectively. Hereinafter, we denote the former as "FPGA-mod-A" and the latter as "FPGA-mod-AB." We did not test the FPGA implementation of mod-B because mod-B utilizes the expanding degrees in the intensification phase which is not suitable for parallel implementation on FPGA as discussed in Section 4.1.

The software implementations were compiled by g++ with -O3 option and were executed on Core-i7 5820K 3.3 GHz with 32 GB main memory. The FPGA implementations were implemented on a XIL-ACCEL-RD-KU115 board comprising a Kintex Ultrascale XCKU115 FPGA (by Xilinx, Inc.) and 16 GB off-chip DDR4-SDRAMs. **Tables 5** and **6** summarize the resource utilizations of the FPGA implementations. The maximum operational frequencies were 90 MHz in FPGA-mod-A and 140 MHz in FPGA-mod-AB. The differences in the operational frequency and resource utilizations will be discussed in Section 6.5.2.

6.2 Evaluation Condition

In each implementation, 50 trials were executed for each graph, respectively, except for C2000.9. As for C2000.9, 10 trials were executed because this graph was considerably time consuming in the evaluation. Each trial was stopped when it either found the best-known solutions or reached 10^9 iterations that were divided into 10^5 restarts (restart per 10^4 iterations).

6.3 Accuracy Evaluation

In order to evaluate accuracy deterioration in our proposed approach, we first compared the number of graphs for which each implementation reached the best-known solutions in all the trials. In this evaluation, we tested the original SBTS, mod-A, mod-B, mod-AB, and FPGA-mod-AB. **Table 7** shows the result of the comparison. As is evident from Table 7, our proposed approach prevented a loss in accuracy in most cases.

Table 8 Average sizes of the cliques obtained for hard instance graphs.

graph	original	mod-A	mod-B	mod-AB	FPGA-mod-AB
brock800_1	22.8	23	22.7	23	22.8
brock800_2	24	24	23.9	24	24
MANN_a45	345	344	345	344.1	344.5
MANN_a81	1,100	1,097	1,100	1,097	1,098.1
C2000.9	78.0	77.0	78.3	77.7	76.9

Table 9 Overall performance gain by algorithm modifications.

		mod-A	mod-B	mod-AB
X_{iter}	best	0.188	0.495	0.192
	avg	1.32	1.04	1.32
	worst	9.22	1.74	10.9
X_{sec}	max	26.1	1.40	24.5
	avg	2.71	1.02	2.46
	min	0.600	0.716	0.500
$X_{iter/sec}$	max	14.7	1.66	14.7
	avg	2.42	1.04	2.33
	min	0.324	0.491	0.209

We then focused on the graphs of which each implementation missed out on reaching the best known solutions in at least one trial. **Table 8** shows a comparison of the average sizes of the obtained cliques for such graphs. Degradations of the solution accuracy were observed for MANN_a45 and MANN_a81 in mod-A, mod-AB, and FPGA-mod-AB. In general, graphs that encode problems from other domains involve the structure derived from their original problems. MANN graphs are from the set covering problems arising from Steiner triple systems [14], and the solution search heuristic in our proposed approach may not be effective for Steiner triple systems so much. Additional evaluations are required to clarify the relationship between the graph structures and effectiveness of our proposed approach.

6.4 Performance Gain by Algorithm Modification

Table 9 represents the performance gain by each variant of our proposed approach over the original SBTS. In Table 9, X_{iter} represents the ratio of iterations to reach the best-known solutions in each variant to those in the original SBTS. Note that when X_{iter} is large, the variant requires more number of iterations than the original SBTS. X_{sec} and $X_{iter/sec}$ represent speedup by total elapsed seconds and the number of iterations per second to reach the best-known solutions, respectively, compared with the original SBTS algorithm. Both of them include the time spent for reading a given graph and generating its complementary graph. Trials that did not obtain the best-known solutions are excluded. Although X_{sec} is affected by the increase/decrease of the number of iterations, $X_{iter/sec}$ does not vary so much regardless of the number of iterations. Therefore, $X_{iter/sec}$ is a better criterion for evaluating performance gain than X_{sec}.

As can be seen in Table 9, mod-A and mod-AB achieve comparable performance. As described in Section 4.1, mod-A and mod-AB do not incur indirect array indexing by referencing the expanding degrees occurring in the original SBTS. This leads to reducing the random access to the main memory and promoting burst access to it, thereby improving the throughput of the search. Furthermore, both of the variants utilize only one parameter, tabu tenure, to select a vertex to be swapped in its intensification phase, whereas the original utilizes two kinds of parameters, the expanding and diversifying degrees. This leads to reducing the frequency

© 2022 Information Processing Society of Japan

Table 10 Performance comparison between FPGA and software implementations.

graph	software (original)			software (mod-AB)			FPGA-mod-AB (140 MHz, P = 64)				
	size	#iter$_{avg}$	sec$_{avg}$	size	#iter$_{avg}$	sec$_{avg}$	size	#iter$_{avg}$	sec$_{avg}$	X_1	X_2
brock400_1	27(27)	1.96×10^7	41.7	27(27)	7.33×10^6	5.58	27(27)	7.16×10^6	9.42	4.43	0.592
brock400_2	29(29)	4.34×10^6	9.19	29(29)	1.55×10^6	1.20	29(29)	1.19×10^6	1.63	5.65	0.740
brock400_3	31(31)	5.00×10^5	1.09	31(31)	3.51×10^5	0.292	31(31)	3.21×10^5	0.487	2.24	0.599
san400_0.7_1	40(40)	3.99×10^5	1.07	40(40)	1.17×10^5	0.177	40(40)	9.89×10^4	0.190	5.65	0.930
brock800_1	23(22.8)	3.48×10^8	4,388	23(23)	3.21×10^8	586	23(22.8)	2.71×10^8	389	11.3	1.51
brock800_2	24(24)	1.22×10^8	1,520	24(24)	1.17×10^8	211	24(24)	1.33×10^8	190	7.98	1.11
brock800_3	25(25)	1.03×10^8	1,292	25(25)	8.06×10^7	147	25(25)	7.12×10^7	102	12.6	1.44
brock800_4	26(26)	2.86×10^7	359	26(26)	3.55×10^7	64.3	26(26)	4.14×10^7	59.4	6.04	1.08
C1000.9	68(68)	2.00×10^6	9.78	68(68)	2.03×10^7	14.5	68(68)	1.83×10^7	25.8	0.378	0.563
san1000	15(15)	1.14×10^6	26.0	15(15)	2.32×10^5	1.06	15(15)	2.77×10^5	0.581	44.7	1.83
MANN_a45	345(345)	5.03×10^6	7.97	345(344.1)	4.88×10^6	7.66	345(344.5)	2.48×10^8	330	0.0242	0.0232
p_hat1500-1	12(12)	1.01×10^5	7.59	12(12)	1.07×10^5	0.952	12(12)	1.07×10^5	0.581	13.1	1.64
C2000.5	16(16)	3.11×10^4	2.77	16(16)	4.50×10^4	0.631	16(16)	3.48×10^4	0.617	4.49	1.02
MANN_a81	1,100(1,100)	7.52×10^5	18.3	1,097(1,097)	-	-	1,100(1,098.1)	7.78×10^7	132	0.139	-
keller6	59(59)	9.61×10^6	411	59(59)	1.38×10^7	49.1	59(59)	4.45×10^6	9.13	45.0	5.38
C4000.5	18(18)	4.90×10^6	844	18(18)	7.62×10^6	89.2	18(18)	6.21×10^6	16.5	51.1	5.40
C2000.9	78(78.0)	-	-	80(77.7)	3.88×10^8	476	78(76.9)	-	-	-	-

of the main memory access in itself. On the other hand, performance gain by mod-B is limited in the most cases. This is reasonable because the calculation of the tabu tenures hardly occupies the total elapsed time as shown in Fig. 3 in Section 3.4. The maximum of $X_{iter/sec}$ is 1.66× in mod-B, which seems to be high for the small occupation ratio of the tabu tenure calculation. To clarify the reason for this, it may be necessary to analyze the difference in behavior at each iteration between the original SBTS and mod-B, which is one of our future tasks. Overall, performance gain by the algorithm modification mainly arises from the changes in the vertex selection heuristic in the intensification phase.

Table 11 Performance of FPGA-mod-A (90 MHz, P = 64).

graph	size	#iter$_{avg}$	sec$_{avg}$	$X_{iter/sec}$	X_{sec}
brock400_1	27 (27)	5.36×10^6	11.4	0.617	0.825
brock400_2	29 (29)	1.44×10^6	3.07	0.642	0.529
brock400_3	31 (31)	3.62×10^5	0.790	0.690	0.611
san400_0.7_1	40 (40)	1.06×10^5	0.269	0.757	0.704
brock800_2	24 (24)	1.72×10^8	393	0.628	0.484
brock800_3	25 (25)	9.21×10^7	210	0.630	0.488
brock800_4	26 (26)	4.01×10^7	91.2	0.630	0.652
C1000.9	68 (68)	1.89×10^7	43.1	0.617	0.599
san1000	15 (15)	2.64×10^5	0.764	0.724	0.760
p_hat1500-1	12 (12)	1.20×10^5	0.696	0.931	0.834
C2000.5	16 (16)	4.76×10^4	0.854	0.987	0.722
keller6	59 (59)	4.36×10^6	13.6	0.661	0.673
C4000.5	18 (18)	5.78×10^6	22.9	0.673	0.722

6.5 Performance Gain by Hardware Acceleration

Firstly, we present the performance comparison of FPGA-mod-AB with the original SBTS and mod-AB. In this evaluation, we focus on the the same graphs as those used for the performance profiling in Section 3.4. In the other 63 graphs, all the software implementations and FPGA-mod-AB reach the best-known solutions in less than 1 second according to our experiments. We exclude such graphs and focus on the remaining 17 ones.

Table 10 shows the performance comparison of each implementation. "size" and sec$_{avg}$ have the same meanings as in Table 3. The time spent on downloading the data into the FPGA is also accounted for by sec$_{avg}$ of the FPGA implementation. #iter$_{avg}$ denotes the average number of iterations to obtain the best-known solutions. X_1 and X_2 represent the speedup values of the FPGA implementation by sec$_{avg}$ compared with the original SBTS and mod-AB, respectively. Trials that did not obtain the best-known solutions are excluded for evaluating X_1 and X_2.

As shown in Table 10, FPGA-mod-AB performs well (up to 51.1× speedup) as compared with the original SBTS. On the other hand, the speedup over mod-AB is limited (up to 5.40×). As discussed in Section 6.4, throughput of mod-AB is superior to that of the original SBTS. This indicates that the speedup of FPGA-mod-AB over the original may result not only from the parallel processing on FPGA but also from the low throughput of the original SBTS on CPU.

6.5.1 Effectiveness of Parallel Processing

As described in Sections 5.3 and 5.4, parallel and pipeline processing accelerate the selection of vertices for swapping and updating of the parameters that occur with the swapping. The inherent parallelism of the former and that of the latter depend on N_v and #adj$_{avg}$, respectively. Overall, both X_1 and X_2 are roughly proportional to N_v and #adj$_{avg}$, respectively, which indicates that FPGA-mod-AB leverages the inherent parallelism.

FPGA-mod-AB outperforms the original SBTS on CPU except for C1000.9, MANN_a45, and MANN_a81. As for C1000.9, mod-AB and FPGA-mod-AB require approximately 10× more iterations than the original SBTS, thereby cancelling out the effectiveness of the parallel and pipeline processing. This may be because mod-AB, i.e., the underlying algorithm of FPGA-mod-AB, causes a loss in accuracy for C1000.9, thereby requiring more iterations to reach the best known solution. While this is also true for MANN graphs, FPGA-mod-AB requires more iterations than mod-AB as well as the original SBTS by orders of magnitude unlike the case of C1000.9. The number of iterations between A and B should be about the same since their underlying algorithm is the same. Investigating why the number of iterations is so different between FPGA-mod-AB and mod-AB for MANN graphs is one of our future tasks.

6.5.2 Effectiveness of Simplifying Tabu Tenure Calculation

Next, we compare the performance of FPGA-mod-A with FPGA-mod-AB, based on which we discuss the effectiveness of simplifying tabu tenure calculation. **Table 11** shows the performance of FPGA-mod-A. X_{sec} shows the ratio of elapsed time of

© 2022 Information Processing Society of Japan

Table 12 Performance of FPGA-mod-AB when off-chip DRAMs unused (133 MHz, P = 64).

graph	size	#iter$_{avg}$	sec$_{avg}$	X_1	X_2	X_3
brock400_1	27 (27)	7.16×10^6	5.25	7.95	1.06	1.79
brock400_2	29 (29)	1.19×10^6	0.941	9.76	1.28	1.73
brock400_3	31 (31)	3.21×10^5	0.296	3.69	0.986	1.65
san400_0.7_1	40 (40)	9.89×10^4	0.130	8.26	1.36	1.46
brock800_1	23 (22.8)	2.71×10^8	227	19.3	2.58	1.72
brock800_2	24 (24)	1.33×10^8	109	14.0	1.94	1.75
brock800_3	25 (25)	7.12×10^7	59.4	21.7	2.47	1.72
brock800_4	26 (26)	4.14×10^7	34.8	10.3	1.85	1.71
C1000.9	68 (68)	1.83×10^7	14.8	0.659	0.981	1.74

FPGA-mod-A to that of FPGA-mod-AB. $X_{iter/sec}$ represents the ratio of iterations per second of FPGA-mod-A to that of FPGA-mod-AB. Other notations have the same meanings as in Table 10.

The performance of FPGA-mod-A is inferior to that of FPGA-mod-AB in all the tested cases because of its lower operational frequency. If FPGA-mod-A could operate at the same frequency as FPGA-mod-AB, performance of both would be nearly equal. For example, $X_{iter/sec}$ and X_{sec} for brock400_1 would become 0.960× and 1.28×, respectively. As can be seen from Tables 5 and 6, utilizations of LUTs and flipflops in FPGA-mod-A are increased by 35% and 24%, respectively, compared with FPGA-mod-AB. This is because the remainder calculators for generating random integers are implemented by LUTs and flipflops in FPGA-mod-A (64 remainder calculators are implemented to calculate up to 64 tabu tenures at once). The operational frequency of FPGA-mod-A is decreased by 36% compared with FPGA-mod-AB, which implies that removing the remainder calculators leads to reducing the critical path delay. Thus, simplification of the tabu tenure calculation contributes to increasing the performance while reducing hardware resource utilization.

6.5.3 Reducing Off-chip DRAM Access Latency

As discussed in Section 6.5, the speedup of FPGA-mod-AB over mod-AB is limited. The access latency of the off-chip DRAMs is considered to be a major factor in limiting the performance. In the FPGA implementation, the list table was assigned to the off-chip DRAMs. As can be seen from Table 6, 1,433 on-chip memories (block RAMs) were still available. By implementing the list table using the block RAMs, the off-chip DRAM access latency can be canceled.

We could implement a circuit that could handle the graphs with 800 vertices by using 256 block RAMs instead of using off-chip DRAMs to implement the list table (C1000.9 could be handled as well because the amount of the list table for C1000.9 was small enough). However, the system clock frequency was reduced to 133 MHz because the timing constraints became more severe due to the increase of the block RAMs. **Table 12** shows the performance of the above-stated circuit. In Table 12, X_3 represents the speedup values of the circuit without using the off-chip DRAMs by sec$_{avg}$ over the FPGA implementation (using off-chip DRAMs), and the other notations have the same meanings as in Table 10. X_3 ranges from 1.46× to 1.79×, which implies that the off-chip DRAM access latency accounted for 32% to 44% of the overall execution time in the tested cases.

Although larger graphs can be handled using more block RAMs to implement the list table, overuse of the block RAMs must be avoided to prevent degradation of the system clock frequency from negating the effect achieved from eliminating off-chip DRAM access latency. According to our experiments, the system clock frequency was down to less than 80 MHz by using 448 block RAMs for the list table. In this case, the cancellation of the off-chip DRAM access latency was neutralized in the most of the tested cases although it could handle the graphs with 2,000 vertices without using off-chip DRAMs.

To eliminate the DRAM access latency using a small number of the block RAMs for larger graphs, a cache memory for the list table may be effective. One concern is that multiple cache lines are necessary to hold long adjacency lists, which requires a complicated cache control logic. However, the DRAM access latency for reading such long lists can be negligible because the time it takes to read long lists is higher than the DRAM access latency. Therefore, caching only short adjacency lists that can be held by a single cache line would be a reasonable approach.

7. Conclusions and Future Work

In this paper, we have described an approach for solving the maximum clique problems using FPGA based on the SBTS algorithm. We modified the SBTS algorithm to discard the indirect array indexing occurring in the original SBTS algorithm thereby facilitating its parallel implementation. We then implemented a circuit based on the modified version of SBTS using FPGA and evaluated the performance. Using the DIMACS benchmark graphs, we showed that the FPGA implementation is up to 5.40× and 51.1× faster than the same algorithm and the original SBTS algorithm, respectively. On the other hand, it has also been implied that the effectiveness of our proposed approach would depend on the structure of the graphs derived from their original problems. In order to clarify this point, we need to investigate the relationship between the structure of graphs and the accuracy of our proposed approach, which will be one of the tasks for future work.

Our current implementation can handle all the graphs in DIMACS benchmark graph suite. However, real-world graphs are considerably larger (comprising more than 1 M vertices) than the DIMACS graphs. To extend the graph size that could be handled, it might be necessary to allocate most of the tables and arrays on the on-chip memories to the off-chip DRAMs, which could cause performance degradation by increasing the memory access latency. To overcome the performance degradation, we need to investigate appropriate data allocation for the on-chip memories and the off-chip DRAMs that could fully utilize the burst access operation of the DRAMs, which is also a task for future work.

Another topic for future work is applying our proposed approach to some variants of the maximum clique problems, e.g., the weighted maximum clique problems. In Refs. [15], [16], parallel algorithms have been proposed for the weighted maximum clique problems. Extending our proposed approach to handle the weighted maximum clique problems and comparing its performance with the existing parallel algorithms are also tasks for future work.

© 2022 Information Processing Society of Japan

References

[1] Pardalos, P.M. and Xue, J.: The maximum clique problem, *Journal of Global Optimization*, Vol.4, No.3, pp.301–328 (1994).

[2] Strickland, D.M., Barnes, E. and Sokol, J.S.: Optimal Protein Structure Alignment Using Maximum Cliques, *Operations Research*, Vol.53, No.3, pp.389–402 (2005).

[3] Corno, F., Prinetto, P. and Reorda, M.S.: Using symbolic techniques to find the maximum clique in very large sparse graphs, *EDTC-1995*, pp.320–324 (1995).

[4] Jin, Y. and Hao, J.: General swap-based multiple neighborhood tabu search for the maximum independent set problem, *Engineering Applications of Artificial Intelligence*, Vol.37, pp.20–33 (2015).

[5] Wu, Q. and Hao, J.: A review on algorithms for maximum clique problems, *European Journal of Operational Research*, Vol.242, No.3, pp.693–709 (2015).

[6] Kanazawa, K.: Accelerating Swap-Based Tabu Search for Solving Maximum Clique Problems on FPGA, *ISPA-2019*, pp.1033–1040 (2019).

[7] Kanazawa, K.: Solving Maximum Clique Problems using FPGA Based on Swap-Based Tabu Search, *HSI-2020*, 7 pages (2021).

[8] Ordońẽz-Guilleń, N.E. and Martińez-Peŕez, I.M.: Heuristic Search Space Generation for Maximum Clique Problem Inspired in Biomolecular Filtering, *Journal of Signal Processing Systems*, Vol.83, No.3, pp.389–400 (2016).

[9] Batsyn, M., Goldengorin, B., Maslov, E. and Pardalos, P.M.: Improvements to MCS algorithm for the maximum clique problem, *Journal of Combinatorial Optimization*, Vol.27, No.2, pp.397–416 (2013).

[10] Tomita, E., Sutani, Y., Takahashi, S. and Wakabayashi, M.: A simple and faster branch-and-bound algorithm for finding a maximum clique, *WALCOM-2010*, pp.191–203 (2010).

[11] Johnson, D.S. and Trick, M.A.: *Cliques, coloring, and satisfiability: second DIMACS implementation challenge*, American Mathematical Society (1996).

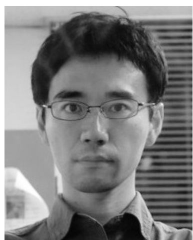
[12] Kanazawa, K. and Cai, S.: FPGA Acceleration to Solve Maximum Clique Problems Encoded into Partial MaxSAT, *MCSoC-2018*, pp.217–224 (2018).

[13] Cai, S., Luo, C., Thornton, J. and Su, K.: Tailoring Local Search for Partial MaxSAT, *AAAI-2014*, pp.2623–2629 (2014).

[14] Mannino, C. and Sassano, A.: Solving hard set covering problems, *Operations Research Letters*, Vol.18, No.1, pp.1–6 (1995).

[15] Baz, D., Hifi, M., Wu, L. and Shi, X.: A parallel ant colony optimization for the maximum-weight clique problem, *IPDPSW-2016*, pp.796–800 (2016).

[16] Sevinc, E. and Dokeroglu, T.: A novel parallel local search algorithm for the maximum vertex weight clique problem in large graphs, *Soft Computing*, Vol.24, No.5, pp.3551–3567 (2020).



Kenji Kanazawa received his Ph.D. from University of Tsukuba in 2012. He is currently an assistant professor at University of Tsukuba. His research interests include parallel processing, reconfigurable computing systems, and highly efficient computing systems using hardware accelerator.

© 2022 Information Processing Society of Japan