Solving Maximum Clique Problems using FPGA Based on Swap-Based Tabu Search

Kenji Kanazawa
Faculty of Engineering, Information and Systems, University of Tsukuba, Japan
kanazawa@cs.tsukuba.ac.jp

Abstract

The Swap-Based Tabu Search (SBTS) is a heuristic algorithm for solving the maximum independent set problems and it 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 has abundance of inherent parallelism, it is difficult to accelerate on hardware due to its solution searching heuristic involving the indirect indexing on array components. In this paper, we show a variant of SBTS that removed the indirect array indexing 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 69.1 times faster than the original SBTS algorithm on CPU.

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 occurring in many practical applications [Pardalos and Xue, 1994], e.g., bioinformatics, and VLSI CADs. 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, a clique in G is equivalent to 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 $\overline{G}$. Fig. 1 displays an example of the maximum clique and the maximum independent set. 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.

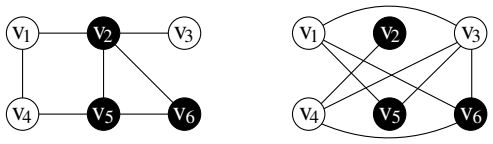

Figure 1: G and its maximum clique (on the left side), and $\overline{G}$ and its maximum independent set (on the right side)

Field-programmable gate array (FPGA) is a programmable LSI, on which very high performance can be achieved by executing a thoroughly parallelized algorithm.

The Swap-Based Tabu Search (SBTS) [Jin and Hao, 2015] is one of the best performing heuristic algorithm for solving the maximum clique problems as well as the maximum independent set problems[Wu and Hao, 2015]. Although SBTS has abundant inherent parallelism, it involves many occurrences of indirect indexing on array components in the form of $X[Y[i]]$ in its solution search heuristic, thereby making its parallel processing difficult.

In this paper, we describe a modified version of the SBTS algorithm that does not require the indirect indexing while maintaining the algorithmic accuracy as that of the original SBTS algorithm and also represent its parallel implementation on FPGA[Kanazawa, 2019]. We then evaluate the performance of our modified SBTS algorithm and its FPGA implementation, based on which we discuss the future tasks.

2 The Swap-based Tabu Search Algorithm

2.1 Definitions of the symbols

Algorithm 1 summarizes the main procedure of the SBTS algorithm. Herein, V and E are the sets of vertices and edges in a graph G, respectively. S denotes an independent set, S_{max} the largest independent set found so far, and f_{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 and more adjacent vertices in S.

k is called the "mapping degree" of the vertex $v \in NS_k$ and v's mapping degree is denoted by $m(v)$. SBTS searches for a solution by iteratively selecting a vertex in any of NS_k and moving it to S and its adjacent vertices in S to any of NS_k, i.e., "swapping" a vertex in any of NS_k and its adjacent

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: Init($S, NS_0, NS_1, NS_2, NS_{>2}, m(), d(), e(), t()$);
2: $S_{max} \leftarrow S$; $f_{max} = |S|$;
3:
4: /* Searching a Solution. */
5: **for** $Iter$ = 1 to $Iters_{max}$ **do**
6: $\dot{v} \leftarrow$ NULL;
7: **if** $NS_0 \neq \emptyset$ or $NS_1 \neq \emptyset$ **then**
8: `/* Intensification Phase */`
9: $\dot{v} \leftarrow$ Sel_Intense($S, NS_0, NS_1, e(), d(), t(), Iter$);
10: **if** $\dot{v} ==$ NULL **then**
11: `/* Diversification Phase */`
12: $\dot{v} \leftarrow$ Sel_Diversify($S, NS_2, NS_{>2}, d(), t(), Iter$);
13: **end if**
14: **end if**
15: $S \leftarrow$ updateIndependentSet($\dot{v}, S, NS_0, NS_1, NS_2, NS_{>2}$);
16: update_parameters($m(), d(), e(), t()$);
17: **if** $|S| > f_{max}$ **then**
18: $S_{max} \leftarrow S$; $f_{max} \leftarrow |S|$;
19: **end if**
20: **end for**
21: **return** S_{max};

vertices in S. Therefore, $m(v) - 1$ is equal to the decrease of $|S|$ when $v \in V\backslash S$ is moved to S.

$e(v)$ and $d(v)$ denote v's "expanding degree" and "diversifying degree", respectively, which are the decision information used in the heuristic of SBTS. $e(v)$ represents the number of v's adjacent vertices in $V\backslash S$ whose mapping degree is equal to 1. Furthermore, $e(v) - 1$ corresponds to the number of vertices whose mapping degree becomes zero if $v \in S$ is moved to $V\backslash S$. Therefore, selecting $v \in V\backslash S$ whose adjacent vertex in S has larger expanding degree and moving it to S leads to reach larger independent set in the next iteration. $d(v)$ indicates the number of its adjacent vertices in $V\backslash S$. As mentioned above, when $v \in V\backslash S$ is moved to S, its adjacent vertices in S also move to any of NS_k in accordance with the changes of their mapping degrees. Therefore, selecting a vertex with a larger diversifying degree leads to more changes in NS_k, thereby diversifying the choices in the next iteration.

$t(v)$ 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.

2.2 Overall procedure

SBTS begins by considering an initial independent set. First, it sets S to empty. Then, until V becomes empty, it iteratively selects $v \in V$ at random, moves v to S, and removes all of v's adjacent vertices from V. Then, the algorithm 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 $NS_{>2}$ in accordance with their mapping degrees. Subsequently, it iteratively alternates the *intensification phase* (i.e., searching a better solution than the current one) and *diversification phase* (i.e., perturb 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. In the following discussion, $\dot{v}$ denotes a vertex in any of NS_k that is selected to move to S at the current iteration. As mentioned in Section 2.1, when $\dot{v}$ is moved to S, its adjacent vertices in S must be moved to $V\backslash S$ at the same time, except that $\dot{v}$ is selected from NS_0. Herein, we denote such vertices as w. The size of 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, the search moves to another solution without deteriorating the size of S. Otherwise, the size of S is decreased.

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

- $\dot{v}$'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'').

Note that w, w', and w'' represent all the corresponding vertices of them, respectively. We will explain how the mapping and diversifying degrees are changed in Section 3.1. Also, the expanding degrees are updated accordingly. Then, the vertices whose mapping degrees have become changed are moved to any of NS_k ($k = 0, 1, 2, >2$) in accordance with their new mapping degrees. Furthermore, the tabu tenures are calculated for w as follows:

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

2.3 Vertex selection heuristics

Intensification phase

The intensification phase is meant 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, $\dot{v}$ 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.

   a. 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.

   b. 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, select the vertex that has the largest diversifying degree among them (ties are broken at random).

If no vertices to be added to S are found in the intensification phase, it is implied that the search has reached a local optima. At this point, SBTS switches the search to the diversification phase.

Diversification phase

The diversification phase is meant for escaping from local optima by perturbing the current solution. In the diversification phase, $\dot{v}$ is selected as follows.

(1) If $|NS_1| > |NS_2| + |NS_{>2}|$, 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 (hereinafter $p = 0.5$), 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.

2.4 Changes to facilitate parallel processing

Modify the heuristic in the intensification phase

The most difficult part to parallelize is Step (2) in the intensification phase. As described in Section 2.3, when $\dot{v}$ is selected from NS_1, it is necessary for each candidate vertex in NS_1 to determine its adjacent vertex in S 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 of a vertex can be read immediately. This requires indirect array indexing in the form of $X[Y[i]]$, where X and Y correspond the expanding degree table and v's adjacency list, respectively, and $Y[i]$ represents one of the v's adjacent vertices in S. If there are multiple vertices that have common adjacent vertex in S, they can cause access conflict in the expanding degree table. It is difficult to schedule the vertices in NS_1 to avoid the access conflict in advance because adjacent vertex in S for each vertex varies in every iteration. By preparing the duplicates of the expanding degree table, the access conflict may be avoided. However, it requires a very complicated control logic on parallel circuit to maintain the consistency between the duplicates.

To solve this problem, we utilize the number of iterations after passing the tabu tenures instead of the expanding degrees so that the vertex selection method can be parallelized more easily. The new vertex selection method in the intensification phase is as follows:

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

(2) 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).**

In this method, Step (2) can be executed by comparing the tabu tenures of the vertices in NS_1 and selecting the vertex with the minimum tabu tenure, which can be parallelized avoiding the indirect array indexing.

Simplify the tabu tenure calculation

To calculate the tabu tenures, it is necessary to calculate $\mathcal{R}(x)$, which is equivalent to remainder of r divided by x. The simple ways to calculate remainder on hardware are the restoring or non-restoring divisions. However, these consume calculation time proportional to the width of the operands. To calculate the tabu tenures quickly with simple hardware implementation, we replace $\mathcal{R}(x)$ to 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=1, return $x-1$. Otherwise, return 0.

$\mathcal{R}'(x)$ can be implemented by only a random bit generator and a multiplexer, which requires considerably less hardware resources and is much faster than $\mathcal{R}(x)$.

3 Hardware implementation

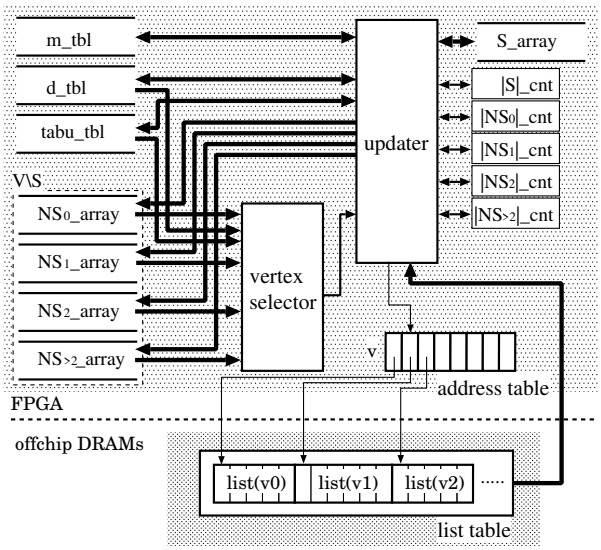

Figure 2: Block diagram of the proposed approach on hardware

Fig. 2 shows a block diagram of the hardware. In Fig. 2, 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 |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. All the tables 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 |NS$_k$|_cnts in each iteration.

The aforementioned tables and arrays, with the exception of the list and address tables, are in the form of the content addressable memory (CAM), in which a vertex number is used as 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, it means that v has moved from NS_k to S at the iteration.

m_tbl, d_tbl, tabu_tbl, S_array, and NS_k_arrays are divided into P sub-banks such that up to P vertices in list(v) are processed at once. Also, |S|_cnt and |NS_k|_cnts comprise P sub-counters, respectively. The i-th sub-counters hold the number of 1 (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 index of 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 $v >> \log_2 P$ of the $(\log_2 P)$-th sub-bank of m_tbl, where '>>' represents the right-shift operation. Each vertex corresponds injectively to a certain sub-bank, which facilitates the sub-bank addressing and parallel processing of multiple vertices.

list(v) whose length L exceeds P is divided into L/P sub-lists and each sub-list is processed in turn. Vertices in each sub-list are placed to the fixed position such that they access to only the corresponding sub-banks of m_tbl, d_tbl, tabu_tbl, and the arrays (i.e., the sub-banks whose indices are equivalent to the lower $\log_2 P$ bits of the vertices).

3.1 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 to 1 in NS_0_array.

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 list($\dot{v}$) and then update m_tbl[w'] and d_tbl[w'] and then set NS_k_array[w'] according to the new values of m_tbl[w'].

iv. Update |S|_cnt and |NS_k|_cnts.

v. Repeat Steps ii) to iv) until all the bits in NS_0_array become 0.

Subsequently, the following procedure is iteratively executed on FPGA to search a solution ($k = 0, 1, 2$ or >2 in the following procedure).

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, and then update m_tbl[$\dot{v}$], and d_tbl[$\dot{v}$].

viii. Read vertices w and w' from list($\dot{v}$), and then update tabu_tbl[w], m_tbl[w], d_tbl[w], m_tbl[w'], and d_tbl[w']. Then, set S_array[w] to 0, and set NS_k_array[w] and NS_k_array[w'] according to the new values of m_tbl[w] and m_tbl[w'].

ix. Read vertices w'' from list(w), update m_tbl[w''] and d_tbl[w''], and then set NS_k_array[w''] according to the new values of m_tbl[w''].

In the above procedure, |S|_cnt and |NS_k|_cnts are also updated accordingly whenever S_ and NS_k_arrays are updated.

Updating tables and arrays

In each iteration, the mapping degrees of w and w' are incremented, and the diversifying degrees of w and w' are decremented. This is because $\dot{v}$, i.e., one of the adjacent vertices of w and w', has moved to S. On the other hand, The mapping degrees of w'' are decremented and the diversifying degrees of w'' are incremented. This is because w', i.e., one of the adjacent vertices of w'', have moved to $V \backslash S$. $\dot{v}$'s mapping and diversifying degree become 0 and the length of list($\dot{v}$), i.e., the number of $\dot{v}$'s adjacent vertices, respectively.

In Step viii, up to P vertices are read from list($\dot{v}$) at once and then their values of the tables and NS_k_arrays are updated accordingly, which is repeated until all the vertices in list($\dot{v}$) are processed. In Step ix, vertices in list(w) are processed likewise. There exist abundant parallelism in Steps viii and ix, depending on the number of adjacent vertices for each vertex. There are no data dependencies in updating these tables and arrays, which is very suitable task for parallel processing.

Selection of a vertex to be swapped

In our proposed approach, the selections of $\dot{v}$ can be categorized into the following three types of methods (note that the intensification phase is changed as described in Section 2.4).

(x) Select a vertex among the vertices ν that satisfy NS_k_array[ν] = 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 ν that satisfy NS_1_array[ν] = 1 (corresponding to Step (2) in the intensification phase).

(z) Select a vertex with the largest value of d_tbl among the vertices ν that satisfy NS_k_array[ν]= 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 selection 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.

4 Performance evaluation

4.1 Experimental setup

We compared our modified SBTS algorithm and its FPGA implementation with the original SBTS algorithm using DIMACS graph suite[Johnson and Trick, 1996]. For the evaluation, we tested two types of software solvers, the original SBTS algorithm published by the developer[1] (hereinafter "original") and our modified SBTS algorithm (hereinafter "modified"), and the FPGA implementation of the modified

[1] http://www.info.univ-angers.fr/ hao/mis.html.

SBTS (hereinafter "FPGA solver"). The software solvers were compiled using g++ with the -O3 option and were executed on a Core-i7 5820K 3.3 GHz processor with 16 GB main memory. The FPGA solver ($P = 64$ and 140 MHz system clock frequency) was implemented on a XIL-ACCEL-RD-KU115 board comprising a Kintex Ultrascale XCKU115 FPGA (Xilinx, Inc.) and 16 GB off-chip DDR4-SDRAMs. Table 1 summarizes the resource utilization of the FPGA solver.

Table 1: Resource utilization of the FPGA solver ($P = 64$)

Resource	Consumed / Total
LUT(logic element of FPGA)	257.3K / 660.8K (38.9%)
flipflop	256.9K / 1321K (19.4%)
36Kb on-chip memory (block RAM)	727 / 2160 (33.7%)

Evaluation condition In each solver, 10 trials were conducted for each graph. Each trial was stopped when it either found the best-known solutions, i.e., the cliques of the graphs with the best-known sizes, or reached 10^9 iterations that were divided into 10^5 restarts (restart per 10^4 iterations).

4.2 Accuracy evaluation

Table 2 shows the comparison of the number of graphs for which each solver found the best known solutions in all the 10 trials. As is evident from Table 2, our proposed approach prevented the loss of accuracy in most cases.

We then focused on the graphs that each solver in at least one trial skipped out on the best-known solutions. Table 3 shows the comparison of the average sizes of the obtained cliques for such graphs. Degradations of the solution quality were observed for MANN_a45 and MANN_a81 in the modified and the FPGA solvers. The solution search heuristic of our proposed approach may not be appropriate for the graph structures in MANN graphs. To make it clear, additional evaluations are required to evaluate the relationship between the graph structures and effectiveness of our proposed approach.

Table 2: The number of graphs for which each solver found the best-known solutions in the 10 trials

# of graphs	original	modified	FPGA solver
80	78	77	77

Table 3: Average sizes of the cliques obtained for hard instance graphs

graph	original	modified	FPGA solver
C2000.9	77.2	**77.7**	76.9
brock800_1	22.6	**23.0**	**23.0**
MANN_a45	**345.0**	344.0	344.5
MANN_a81	**1100.0**	1097.0	1098.0

4.3 Performance gain

Performance gain by modifying the SBTS algorithm

We first evaluated the performance of the software solvers. The modified reached the best-known solutions faster than the original in 69 graphs. The number of iterations per second in the modified was $2.51\times$ on average and $15.6\times$ at maximum as compared with the original, which indicates that the throughput of the search became improved.

This may result in the difference of the intensification phases in the original and the modified solvers. As described in Section 2.4, the modified does not incur the indirect array indexing by the reference to the expanding degrees occurring in the original. 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, the modified utilizes 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 of the main memory access in itself.

Performance gain by hardware acceleration

Next, we compared the performance of the FPGA solver with the above-stated two types of software solvers. According to our experiments, all the software solvers and the FPGA solver reached the best-known solutions for 63 graphs among the 80 graphs in less than 1 second. In the following evaluation, we exclude such graphs and focus on the remaining 17 ones.

Table 4 shows the profiles of such benchmark graphs. In Table 4, N_v and N_E denote the number of vertices and edges in a graph; $\#adj_{avg}$ shows the average number of adjacent vertices of each vertex, i.e., the average length of list(v), "best" denotes the best-known size of the maximum cliques; and "density" denotes the graph densities, i.e., the actual number of edges over the maximum possible number of edges $(= 2 \times N_E/(N_v(N_v - 1)))$.

Table 5 shows the performance comparison of each solver. The graph names are abbreviated in Table 5. "size" denotes the best sizes of the obtained cliques in the experiments (the average sizes are in brackets). "#avg.iter" and "avg.sec" denote the average number of iterations and the average execution time in seconds, respectively, to obtain the best-known solutions. "avg.sec" also includes the initialization time (the time spent for reading the instances and generating the complementary graphs). The time spent on downloading the data into the FPGA is also accounted for by avg.sec of the FPGA solver. X_1 and X_2 represent the speedup values of the FPGA solver by avg.sec compared with the original and modified, respectively. The trials that did not obtain the best-known solutions are excluded for evaluating X_1 and X_2.

As shown in Table 5, the FPGA solver achieves well (up to $69.1\times$ speedup) as compared with the original, and on the other hand, the speedup over the modified is limited (up to $5.84\times$). This indicates that our approach to facilitate parallel processing on FPGA contributes to promote the throughput of the algorithm on CPU as well. In other words, the speedup of

Table 4: Benchmark profile

graph	N_V	N_E	$\#adj_{avg}$	density(%)	best
brock400_1	400	59723	100.4	74.8	27
brock400_2	400	59786	100.1	74.9	29
san400_0.7_1	400	55860	119.7	70.0	40
brock800_1	800	207505	280.2	64.9	23
brock800_2	800	208166	278.6	65.1	24
brock800_3	800	207333	280.7	64.9	25
brock800_4	800	207643	279.9	65.0	26
C1000.9	1000	450079	98.8	90.1	68
DSJC1000	1000	499652	499.3	50.0	15
san1000	1000	250500	498.0	50.2	15
MANN_a45	1035	533115	3.8	99.6	345
p_hat1500-1	1500	284923	1119.1	25.3	12
C2000.5	2000	999836	999.2	50.0	16
C2000.9	2000	1799532	199.5	90.0	80
MANN_a81	3321	5506380	3.9	99.9	1100
keller6	3361	4619898	610.9	81.8	59
C4000.5	4000	4000268	1998.9	50.0	18

Table 5: Performance comparison

graph	software (original) size	#avg.iter	avg.sec	software (modified) size	#avg.iter	avg.sec	FPGA solver (P = 64) size	#avg.iter	avg.sec	X_1	X_2
br400_1	27 (27)	1.39×10^7	27.5	27 (27)	4.94×10^6	3.76	27 (27)	4.96×10^6	6.56	4.20	0.574
br400_2	29 (29)	4.88×10^6	9.80	29 (29)	1.79×10^6	1.38	29 (29)	1.23×10^6	1.69	5.81	0.816
s_0.7_1	40 (40)	4.15×10^5	1.087	40 (40)	7.52×10^4	0.0933	40 (40)	1.91×10^5	0.315	3.45	0.296
br800_1	23 (22.8)	4.09×10^8	4700	23 (23)	3.18×10^8	568	23 (23)	3.43×10^8	496	9.47	1.15
br800_2	24 (24)	1.63×10^8	1869	24 (24)	1.17×10^8	202	24 (24)	1.36×10^8	195	9.58	1.04
br800_3	25 (25)	1.01×10^8	1161	25 (25)	6.44×10^7	113	25 (25)	1.27×10^8	180	6.45	0.626
br800_4	26 (26)	2.07×10^7	240	26 (26)	2.53×10^7	45.3	26 (26)	3.38×10^7	47.7	5.02	0.948
C1000.9	68 (68)	1.83×10^6	8.35	68 (68)	2.03×10^7	14.4	68 (68)	1.15×10^7	16.4	0.510	0.880
DSJC	15 (15)	3.44×10^4	1.052	15 (15)	7.23×10^4	0.385	15 (15)	8.31×10^4	0.323	3.25	1.19
s1000	15 (15)	1.32×10^6	28.6	15 (15)	2.40×10^5	1.057	15 (15)	1.75×10^5	0.415	69.1	2.55
a45	345 (345)	3.33×10^6	5.89	344 (344)	-	-	345 (344.5)	8.48×10^7	112	0.0526	-
p1500-1	12 (12)	9.42×10^4	6.64	12 (12)	1.15×10^5	1.032	12 (12)	7.97×10^4	0.540	12.3	1.91
C2000.5	16 (16)	2.82×10^4	2.11	16 (16)	7.11×10^4	0.523	16 (16)	4.68×10^4	0.602	3.51	0.868
C2000.9	78 (77.2)	-	-	80 (77.7)	3.88×10^8	476	78 (76.9)	-	-	-	-
a81	1100 (1100)	7.60×10^5	18.9	1097 (1097)	-	-	1098 (1098)	-	-	-	-
keller6	59 (59)	6.82×10^6	287	59 (59)	1.68×10^7	58.4	59 (59)	4.88×10^6	10.0	28.7	5.84
C4000.5	18 (18)	4.47×10^6	786	18 (18)	6.80×10^6	76.8	18 (18)	6.98×10^6	18.3	42.9	4.19

the FPGA solver over the original may result not only from the parallel processing but also from the low throughput of the original.

Reducing off-chip DRAM access latency As discussed above, the speedup of the FPGA solver over the modified is limited. The access latency of the off-chip DRAMs is considered to be a major factor in limiting the performance of a search. In the FPGA solver, the list table was assigned to the off-chip DRAMs. As can be seen from Table 1, 1433 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. However, the system clock frequency was reduced to 133 MHz because the timing constraints became harder by the increase of the block RAMs. Table 6 shows the performance of the above-stated circuit. In Table 6, X_3 represents the speedup values of the circuit without using the off-chip DRAMs by avg.sec over the FPGA solver (with using off-chip DRAMs), and the other notations have the same meanings as in Table 5. X_3 ranges from 1.54× to 1.78×, which implies that the off-chip DRAM access latency accounted for $35\% - 44\%$ 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 so that the system clock frequency degradation does not neutralize the cancellation of the off-chip DRAM access latency. To cancel 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 of the concerns of doing this 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 reasonable.

Table 6: Performance of the FPGA solver ($P = 64$, without using off-chip DRAMs, and 133 MHz system clock frequency)

graph	size	#avg.iter	avg.sec	X_1	X_2	X_3
b400_1	27 (27)	4.96×10^6	3.69	7.47	1.02	1.78
b400_2	29 (29)	1.23×10^6	0.953	10.3	1.45	1.77
b800_1	23 (23)	3.43×10^8	288	16.3	1.97	1.72
b800_2	24 (24)	1.36×10^8	115	16.3	1.77	1.70
b800_3	25 (25)	1.27×10^8	106	10.9	1.06	1.69
b800_4	26 (26)	3.38×10^7	28.6	8.38	1.58	1.67
400_0.7_1	40 (40)	1.91×10^5	0.205	5.29	0.454	1.54

5 Conclusions and future work

In this paper, we 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.

One of the future work is extending the size of the graphs that can be handled. Our current implementation can handle the graphs with 4096 vertices. In our approach, the utilization of the block RAMs was proportional to N_v, which was the main factor in limiting the graph size that could be handled. Considering the utilization of the block RAMs in our current implementation, our proposed approach may handle the graphs with 10K vertices using the same FPGA board. However, real-world graphs are considerably larger (comprising more than 1M vertices) than that. To handle such graphs, it is necessary to move the most part of the tables on the block RAMs to the off-chip DRAMs, which may cause performance degradation by the increase of the DRAM access latency. To reduce the overhead by the increase of the DRAM access latency, we need to investigate appropriate data partition for the block RAMs and the off-chip DRAMs that could fully utilize the bandwidth of the DRAMs as well as introducing the cache for the DRAMs by utilizing the block RAMs.

Another future work is applying our proposed approach to some variants of the maximum clique problems, e.g., the weighted maximum clique problems. In [El Baz *et al.*, 2016; Sevinc and Dokeroglu, 2020], parallel algorithms have been proposed for the weighted maximum clique problems. Our proposed approach would also be applied to solving the weighted maximum clique problems by changing the definitions of the mapping and diversifying degrees, respectively. Extending our proposed approach to handle the weighted maximum clique problems and comparing its performance with the existing parallel algorithms are also our future tasks.

References

[El Baz *et al.*, 2016] Didier El Baz, Mhand Hifi, Lei Wu, and Xiaochuan Shi. A parallel ant colony optimization for the maximum-weight clique problem. In *Proceedings of IPDPSW-2016*, pages 796–800, 2016.

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

[Johnson and Trick, 1996] D. S. Johnson and M. A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

[Kanazawa, 2019] K. Kanazawa. Accelerating swap-based tabu search for solving maximum clique problems on fpga. In *Proceedings of ISPA-2019*, pages 1033–1040, 2019.

[Pardalos and Xue, 1994] P. M. Pardalos and J. Xue. The maximum clique problem. *Journal of global Optimization*, 4(3):301–328, 1994.

[Sevinc and Dokeroglu, 2020] Ender Sevinc and Tansel Dokeroglu. A novel parallel local search algorithm for the maximum vertex weight clique problem in large graphs. *Soft Computing*, 24(5):3551–3567, 2020.

[Wu and Hao, 2015] Q. Wu and J. Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.