Comparing Rank Methods in HEFT Scheduling

R. W. Bunney
email: 21298244@student.uwa.edu.au

May 2017

Abstract

The task scheduling problem has been demonstrated to be NP-Complete in both general and restricted cases, which has resulted in a large body of literature that details the variety of efficient approaches to creating good quality schedules. The Heterogeneous Earliest-Finish-Time (HEFT) algorithm is a popular list-scheduling algorithm that has achieved positive response for its good quality schedules on heterogeneous system. Central to the HEFT algorithm is the assignment of an upward rank to each task. We provide a comparison study that shows that the ranking sort is an improvement over a standard topological sort, and that existing approaches to calculating the rank do not necessarily provide the best schedule.

Keywords: Scheduling, Workflow, Distributed computing.

1 Introduction

The scheduling problem is a well-established and extensively-researched topic within computer science [5, 6, 10, 14]. Applications of scheduling include project workflow scheduling, air-traffic control, and big data processing – to name a few [18]. Scheduling has been found to be an NP-Complete problem, in the general sense; as a result, most research is dedicated to developing different heuristics to determine schedules for particular problems. In the case of multi-processor scheduling problems, in which the problem is focused on scheduling computational tasks onto many processors, the aim of the algorithm is to reduce the time it takes for all the tasks to be completed, and utilise the resources available as effectively as possible [8]. Some approaches work on the assumption that system resources are homogeneous [5, 9]; others attempt to utilise the extra resources that might be available on heterogeneous processors [10, 15]. The Heterogeneous Earliest-Finish-Time (HEFT) algorithm is a popular scheduling heuristic that has achieved a lot of attention for its performance and schedule quality [1]. HEFT is split into two parts; task-prioritisation and processor selection. The first part, task prioritisation is of particular interest to those interested in improving the quality of the resultant schedule. The prioritisation method used in HEFT sorts each task in order of priority generated by a ranking heuristic, before they are then assigned to a processor. This ranking method retains the precedence requirements of each task, and can be demonstrated to be a topological order of the task graph [15].

The original HEFT paper provides no justification for why its ranking method provides a better order of tasks for computation than standard a topological sort would; it also fails to validate why the ranking function uses an average of computation costs associated with a given task, over other methods (e.g., maximum or minimum costs). We believe this is is an important oversight; if one is interested in improving the effectiveness of HEFT for various applications[1, 11], it is necessary to understand why certain methods perform better than others within the algorithm, if they do at all. We aim to demonstrate that, firstly, the ranking method provides a better schedule than that of a topological sort; and secondly, there is no significant benefit to using the average of computation costs. It is posited that when compared to a non-unique topological sort, the ranking method produces a shorter schedule length; and that the mean computation cost does not always provide a better schedule when compared against other values, such as maximum and minimum costs for a given task.

When testing the effectiveness of the HEFT methods identified above, we base our assessment on two criteria; firstly, the length of the resultant schedule, referred to as the *makespan* in the literature [16]; and secondly, the execution time of the algorithm. The experiments are conducted over a range of randomly generated graphs of different sizes. The comparison shows that both the makespan and the execution time length of the HEFT algorithm is significantly lower when using the Rank heuristic demonstrated in [15]. We also provide evidence that suggests the average-cost metric performs no better than using values such as the maximum or minimum computation cost of a task.

The remainder of this paper is as follows: In the next section, the literature surrounding Task scheduling is summarised, and motivations for testing HEFT presented. In Section 3, the scheduling problem is formally defined, and an outline of the HEFT algorithm, ranking method, and processors selection presented. Section 4 provides detail on the implementation of the algorithm and the testing environment, and Section 5 presents a comparative study between the ranking and topological sorting approaches. In Section 6, we summarise our results and discuss the implications for future research.

2 Previous and related work

Scheduling problems are, in the general sense, NP-complete [3, 6, 10, 14, 16]. Kwok & Ahmad [10] note that only three sub-problems have polynomial time algorithms. As a result, algorithms and systems that aim to provide schedules for tasks must experiment with different methods that will find the closest-to-optimal makespan for a given set of tasks.

The scheduling problem is represented in the literature as a Directed Acyclic Graph (DAG) [5, 6, 16, 10, 6]. A sequence of tasks may have precedence constraints in which some tasks cannot start before others have completed, and in schedules with a definitive start and ending, tasks do not form loops [9]. The scheduling problem is formally defined in Section 3. In the follow sections, we summarise the areas of research that relate to the HEFT algorithm, and where it sits in the body of literature.

2.1 Homogeneous vs. Heterogeneous Algorithms

As identified in Kwok & Ahmad [10], research on scheduling has focused on scheduling tasks in homogeneous systems. We define a homogeneous system as a system in which computational capacity of resources in the system is equal across the board; this could include homogeneous processors on a CPU; or a HPC grid in which allocated resources are identical in capacity [10]. Early work on scheduling included focus on operating system and process scheduling for hardware [12], which are simplified to homogeneous systems. Seminal approaches to scheduling include the partitioning and scheduling model outlined by Sarkar [14], in which graphs are initially partitioned into effective node 'clusters' and then scheduled based on the partitions. As identified in [5], this load-balancing technique favours homogeneous systems, as the complexity of partitioning a homogeneous graph is already a challenging problem [16].

As a result of the complexity of heterogeneous scheduling problems [6], less focus has been dedicated to heterogeneous research [10], and methods that are present in the literature are confronted by high scheduling costs [15]. HEFT has emerged as one of the most used techniques for scheduling on heterogeneous systems. Comparison studies against over 20 other algorithms have resulted in HEFT being described as one of the best scheduling techniques for schedule length [4] on heterogeneous systems.

2.2 Scheduling Techniques

The literature presents four common techniques for approaching the DAG scheduling problem; list scheduling, clustering, task duplication, and guided random search [15]. Of these, list scheduling is the most common due to its simple approach of allocating a list tasks to processors [10] and their generally superior performance [17]. Clustering algorithms are analogous to load balancing partitions in scheduling, and are typically intended for systems with an unbounded number of resources; this makes it less useful when applied to heterogeneous systems [5, 12]. Task duplication methods have been demonstrated as impractical for both homogeneous and heterogeneous contexts, due to their high complexity [10]. The most popular approach to random search methods is the use of Genetic Algorithms (GA). These have been shown to develop schedules of equal or better quality to that of a list scheduling heuristic [7]; however, they are let down by high execution times that outweigh the increase in schedule quality [15].

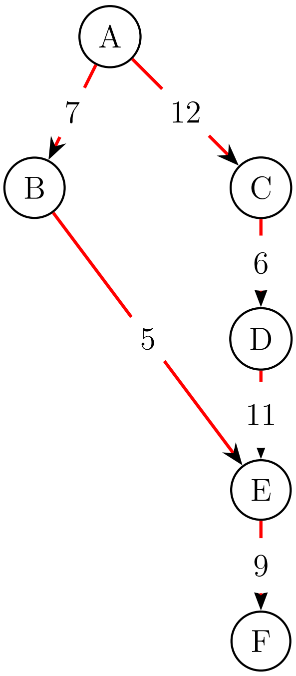
HEFT is a list-scheduling algorithm; it sorts the tasks into a list based on the ranking heuristic mentioned in Section 1, and then allocates tasks to processors according to their position in the list. Methods that extend on HEFT have utilised approaches such as guided search techniques to extend the schedule quality by looking ahead at decisions made by the algorithm [2]; however, these are used on top of the existing list-scheduling framework.

The literature demonstrates the efficacy of HEFT as a scheduling algorithm of good quality schedules. Since its introduction, researchers looking to develop new approaches to scheduling tasks on heterogeneous systems have built upon the algorithm with a variety of approaches, including look-ahead search and run-time scheduling [11, 2]. The combination of good schedule times and the simplicity of a list scheduling algorithm [10] has resulted in HEFT becoming a well-established approach to scheduling on heterogeneous systems.

3 Methodology

3.1 Representing the DAG Scheduling problem

As mentioned in Section 2, the Task Scheduling-problem can be represented formally as a graph G=(V,E), in which V is set of v nodes ands E is a set of e edges [10]. Edge weights on the graph represent communication costs between vertices, and vertex weights represent the time it takes for a task to execute. In this paper, we will use the term 'tasks' or 'nodes' interchangeably to describe vertices, following from the literature [15]. Figure 1 shows an example DAG that represents a workflow schedule, and computation costs for a heterogeneous system with two processors. The edge weights between nodes on the graph represent communication costs between nodes when transferring from one processor to another; this reflects data sizes and transfer rates from each respective task. Data transfer rates on the same processor are assumed to be negligible [9, 15, 13].



Computation Costs

Node	P1	P2
A	7	5
B	8	9
C	4	16
D	2	1
E	8	5
F	12	11

Figure 1: Example DAG and Attributes

3.2 Heterogeneous Earliest-Finish-Time

The HEFT algorithm is a list-scheduling algorithm that is separated into two phases: task prioritisation, and processor selection. The task prioritisation phase utilises *a priori* information about the tasks – computation, and communication costs; task precedence – to give the tasks a priority, or 'rank', by which they are then sorted. The processor selection phase uses this sorted list to assign tasks to processors based the processors capacity; this phase is also called the insertion policy. The HEFT algorithm is summarised in Algorithm 1:

Algorithm 1 The HEFT algorithm

```
1: procedure HEFT_SCHEDULE
2:     Calculate mean computation and communication costs
3:     Calculate upward rank for each task
4:     Sort tasks by decreasing order of rank
5:     for Each task in the sorted list of tasks do
6:         Assign the task to a processor using the insertion policy
```

3.3 Upward Rank

The upward rank is computed by ranking the task graph upward, from the last task (or exit node). The ranking function, $rank_u$, is recursively applied to each node in the graph. The function is described in Equation 1:

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)) \tag{1}$$

Where $succ(n_i)$ refers to the immediate successors of task n_i; for example, in Figure 1, if n_i was A, $succ(n_i)$ would be nodes B and C. $\overline{w_i}$ and $\overline{c_{i,j}}$ refer to the average computation and communication costs for a given node and (i, j) edge, respectively. Table 1 shows the results of applying the upward rank method to the workflow described in Figure 1.

It can be inferred from the table that the final sorted list of nodes does not match the cardinal order of nodes, but does retain precedence constraints; sorting by descending rank gives {A,C,B,D,E,F}.

4 Implementation

All code used to implement and test the HEFT algorithm was written in the Python programming language, with assistance from the popular graph library NetworkX [1]. To assist testing during the implementation, a random DAG generator was developed to create a graph of a given number of nodes

[1] https://networkx.github.io

Table 1: Results of Ranking Heuristic for DAG in Figure 1

Node	$rank_u$
A	71
B	39
C	53
D	38
E	26
F	11

and edges; this was used in conjunction with functions that create a random computation matrix for a given number of processors, and a random communication matrix for the nodes in the graph [2]. In order to determine the differences between the rank and topological sorts, schedules and algorithmic execution times were collected through a loop that generated a number of different graphs with increasing number of nodes. The experimental settings were as follows:

- The number of nodes (N) started at 10 nodes, with an upper bound of 1000 nodes
- For each iteration of nodes, the maximum number of edges the graph could have was 2*N
- The step between each iteration was 50 nodes

Experiments were conducted by generating graphs in a loop, within the boundaries listed above. For each iteration, the task nodes were ranked by both the rank sort method, and a standard topological sort provided by the NetworkX library. The time data collected covered the execution time of the entire algorithm (ranking and processor selection).

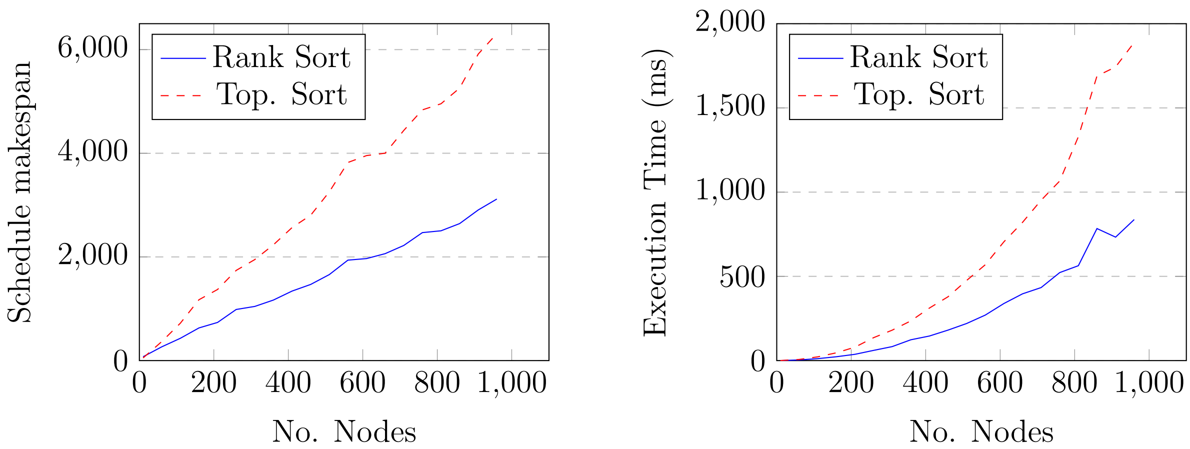
For the first set of tests, initial results were gathered to compare the Rank sort with the topological sort. Then, we ran more simulations that increased the node-to-edge weight of the graph (representing denser, more inter-related task graphs), to determine the impact a more edge-clustered graph would have on the makespan, given the number of nodes had not changed. For the second set of tests, we compared the average-costs ranking

[2] All code used to generate experimental data can be found at https://github.com/myxie/heft

method to a ranking phase that only took into account either the maximum or minimum computation cost of a task. This was run over a number of iterations, from which we calculated the number of better schedules that the average-cost rank produced. This reflects analysis conducted in the original HEFT paper [15].

5 Results

Results from the first set pf experiments are found in Figure 2. Figure 2a demonstrates the difference between the final schedule length of the task graphs when using the $rank_u$ method, or a topological sort. The plot indicates that the $rank_u$ sorting method results in a lower schedule length than when sorted topologically. It appears that the rate at which the length increases as the number of nodes increases is much higher for the topological sort than the ranking sort. There can be more than one topological sort for a given DAG; as the number of nodes in a graph increase, the probability of the topological sort returning a less-than-ideal list of tasks also increases. This would offer an explanation for why the makespan length increases at a higher rate for the topological sort, as the rank sort is only ever generating a unique sort that has an artificial upper-bound by factoring in costs that the topological sort does not.

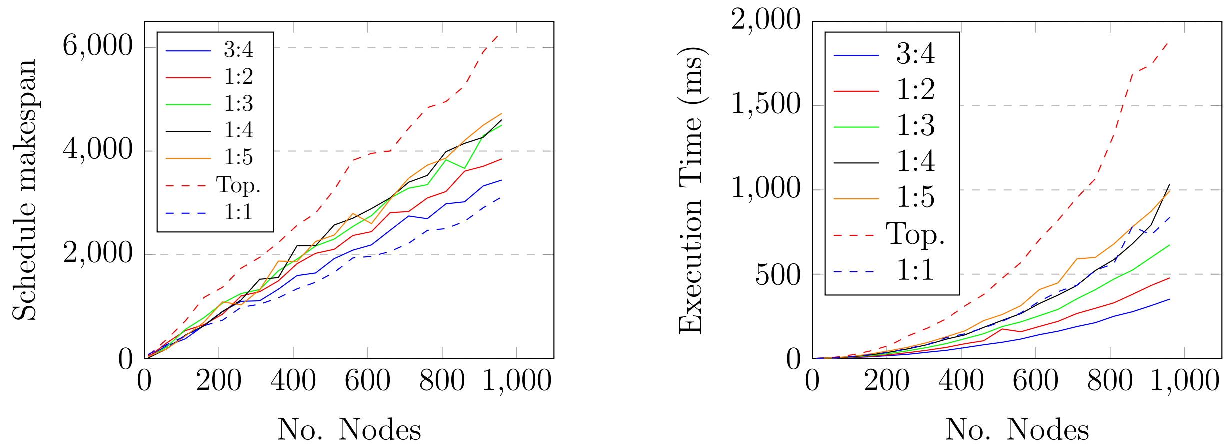


(a) Comparison of Schedule *makespan*

(b) Comparison of schedule execution time

Figure 2: Comparison of Sorting methods in HEFT

Figure 2b shows the difference between the execution time of the whole algorithm for a topologically sorted list of tasks, and a rank sorted list of tasks. It is clear from the plot that the $rank_u$ sort provides a faster execution time for an increasing number of nodes. The higher execution time of the topological sort is most likely due to more time spent looping through processors during the insertion policy, as the list of nodes are not sorted based on their computation costs, and as such the time windows are not as optimally distributed between processors. Of note is the spike in execution time that occurs at 860 nodes; this is likely due to the random graph generator creating a graph that has more potential time windows to allocate as a result of edges in the graph, as it occurs in both the topological sort and rank sort data, and the following data points continue the previous growth trend.



(a) Comparing Node:Edge Ratios and *makespan*

(b) Comparison of Node:Edge Ratios and schedule execution time

Figure 3: Effect of increased Edges on Sorting methods in HEFT

We build up these potential explanations through results in Figure 3. It can be seen that as the number of edges increase in the graph, so does the makespan and schedule times. This is interesting, because the number of nodes does not change between each iteration, just the edges. As the number of edges increase, the more predecessors and successors each task has. The ranking heuristic takes into account each tasks successors, which means as the number of edges increases, more successors are used in the calculation of the rank. It is possible that this 'smooths-out' the rank value of tasks, as there are less distinctive rank values to chose from. Similarly, the noticeable increase in execution time is probably explained by this increase in predecessors. When tasks are being allocated to processors, preceding tasks are used to establish the estimated start time [15]; more edges will result in more predecessors, and thus increase the number of iterations that occur during processor selection. However, this increased edge-density still provides a better schedule and execution time than the original topological sort, which suggests that even though the rank creates less-optimal schedules for increasing edges, it is still better than a topological sort.

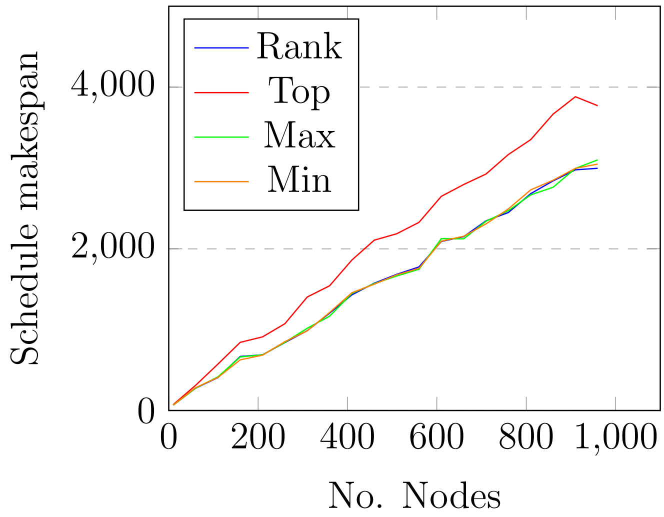


Figure 4: Comparison of ranking costs and *makespan*

Figure 4 provides a demonstration of results from the second lot of tests we ran, which compared using the original average-cost metric to using minimum or maximum values. We can see that there is no clear value that produces the best schedule overall - although all appear to produce a better schedule than the topological sort. To extend our analysis of the results, Table 2 shows a breakdown of the percentage of better schedules that were produced by the original rank. For 98% of schedules, the rank sort was better than a topological sort; however, it only provided better schedules for the minimum computation cost value 41% of the time. What this means is that within this experimental environment, the average-computation cost value provides a worse schedule, on average, than using another metric. Similarly, it only produces a better schedule than the maximum cost half the time.

6 Conclusion

In this paper, we presented a comparison between two ranking methods utilised in the HEFT scheduling algorithm; the original upward rank method,

Table 2: Comparison of Better Schedules

	Topological	Max. Cost	Min. Cost
% of better schedules produced by Mean	98	53	41

$rank_u$; and a standard topological sort. We also compared the cost-metric used in the original HEFT algorithm, the average computation cost on processors, to other cost values. Based on an initial experimental analysis run over randomly generated graphs of increasing sizes, results indicate the ranking sort provides both a better quality schedule, and a faster execution time, when compared against a topological sort. We also have shown that the average-cost metric does not always provide the best schedule, sometimes performing worse than other values. As a result, further research into improving the HEFT algorithm can focus on experimenting with cost values in the ranking method, and developing methods that provide a more consistent result. Planned research involves integrating the HEFT into a distributed scheduling system for large-scale scientific data processing [18]; understanding that the impact the quality of the task-ranks has on the final schedule of the system is pivotal in ensuring the appropriate information is gathered for scheduling in a high-performance computing environment.

References

[1] Arabnejad, H., and Barbosa, J. G. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems 25*, 3 (March 2014), 682–694.

[2] Bittencourt, L. F., Sakellariou, R., and Madeira, E. R. M. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing* (Feb 2010), pp. 27–34.

[3] Blazewicz, J., Lenstra, J., and Kan, A. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics 5*, 1 (1983), 11 – 24.

[4] Canon, Louis-Claude and Jeannot, Emmanuel and Sakellariou, Rizos and Zheng, Wei", editor="Gorlatch, Sergei and Fragopoulou, Paraskevi and Priol, Thierry. *Comparative*

Evaluation Of The Robustness Of DAG Scheduling Heuristics. Springer US, Boston, MA, 2008, pp. 73–84.

[5] Casavant, T. L., and Kuhl, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering 14*, 2 (Feb 1988), 141–154.

[6] Chaudhary, V., and Aggarwal, J. K. A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Parallel and Distributed Systems 4*, 3 (Mar 1993), 328–346.

[7] Hwang, R., Gen, M., and Katayama, H. A comparison of multiprocessor task scheduling algorithms with communication costs. *Computers & Operations Research 35*, 3 (2008), 976 – 993. Part Special Issue: New Trends in Locational Analysis.

[8] Jing-Chiou Liou and M. A. Palis. A Comparison of General Approaches to Multiprocessor Scheduling. In *Proceedings 11th International Parallel Processing Symposium* (Apr 1997), pp. 152–156.

[9] Kwok, Y.-K., and Ahmad, I. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing 59*, 3 (1999), 381 – 422.

[10] Kwok, Y.-K., and Ahmad, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv. 31*, 4 (Dec. 1999), 406–471.

[11] Lee, L. T., Chang, H. Y., Liu, K. Y., Chang, G. M., and Lien, C. C. A dynamic scheduling algorithm in heterogeneous computing environments. In *2006 International Symposium on Communications and Information Technologies* (Oct 2006), pp. 313–318.

[12] Michell, G. D., and Gupta, R. K. Hardware/software co-design. *Proceedings of the IEEE 85*, 3 (Mar 1997), 349–365.

[13] Nikhil, R., et al. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on computers 39*, 3 (1990), 300–318.

[14] Sarkar, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman Publishing, London, UK, 1989.

[15] Topcuoglu, H., Hariri, S., and Wu, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems 13*, 3 (Mar 2002), 260–274.

[16] Ullman, J. D. Np-complete scheduling problems. *J. Comput. Syst. Sci. 10*, 3 (June 1975), 384–393.

[17] Vijayalakshmi, S R; Padmavathi, G. A performance study of ga and lsh in multiprocessor job scheduling. *International Journal of Computer Science Issues 7*, 1 (2010), 37–42.

[18] Wu, C., Tobar, R., Vinsen, K., Wicenec, A., Pallot, D., Lao, B., Wang, R., An, T., Boulton, M., Cooper, I., Dodson, R., Dolensky, M., Mei, Y., and Wang, F. DALiuGE: A Graph Execution Framework for Harnessing the Astronomical Data Deluge. *ArXiv e-prints* (Feb. 2017).