Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking*

J. Barnat[1], L. Brim[1], P. Šimeček[1], and M. Weber[2]

[1] Masaryk University Brno, Czech Republic
[2] University of Twente, The Netherlands

Abstract. Revisiting resistant graph algorithms are those, whose correctness is not vulnerable to repeated edge exploration. Revisiting resistant I/O efficient graph algorithms exhibit considerable speed-up in practice in comparison to non-revisiting resistant algorithms. In the paper we present a new revisiting resistant I/O efficient LTL model checking algorithm. We analyze its theoretical I/O complexity and we experimentally compare its performance to already existing I/O efficient LTL model checking algorithms.

1 Introduction

Model checking real-life industrial systems is a memory demanding and computation intensive task. Utilizing the increase of computational resources available for the verification process is indispensable to handle these complex systems. The three major approaches to gain more computational power include the usage of parallel computers, clusters of workstations and the usage of external memory devices (hard disks), as well as their combination.

In this paper, we focus on external memory devices, where the goal is to develop algorithms that reduce the number of I/O operations an algorithm has to perform to complete its task. This is because the access to information stored on an external device is orders of magnitude slower than the access to information stored in main memory. Thus, the complexity of I/O efficient algorithms is measured in the number of I/O operations [1].

The automata-theoretic approach [2] to model checking finite-state systems against linear-time temporal logic (LTL) reduces to the detection of reachable accepting cycles in a directed graph. Recently, two I/O efficient LTL model-checking algorithms that allow verification of both safety and liveness properties have been proposed in [3] and in [4]. Both algorithms build on breadth-first traversal through the graph and employ the *delayed duplicate detection* technique [5,6,7,8]. The traversal procedure has to maintain a set of visited vertices (*closed set*) to prevent their re-exploration. Since the graphs are large, the closed set cannot be kept completely in main memory. Most of it is stored on an external memory device. When a new vertex is generated (into the *open set*) it is checked against the closed set to avoid its re-exploration. The idea of the delayed duplicate detection technique is to postpone the individual checks and perform them

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

together in a group, for the price of a single scan operation. We assume that the delayed vertices are stored in the main memory as *candidate set*. In order to minimize the number of scan operations which merge the closed set on disk with the candidate set, it is important that the candidate set is as large as possible. In the case of BFS traversal, candidate sets are formed typically from a single BFS level. However, if the level is small, the utility of delaying the duplicate check drops down. A possible solution is to maximize the size of the candidate set by exploring more BFS levels at once. This, in general, leads to revisiting of vertices due to cycles and might violate the correctness of the algorithm. Whether correctness is preserved depends on the algorithm itself. E.g., if an algorithm uses BFS to traverse the reachable part of a graph, revisiting of vertices does not disturb its correctness, while the algorithm for computing a topological sort is not resistant to such revisits.

It is important to note that even though vertex revisits result in performing more (cheap) RAM operations, it might significantly reduce the number of expensive I/O operations. Thus, *revisiting resistant* algorithms are expected to be more I/O efficient than non-resistant ones in practice. In the first part of the paper we explore the notion of a revisiting resistant graph algorithm in more detail.

We are interested in LTL model-checking algorithms for very large *implicit graphs*, i.e., graphs defined by an initial vertex and a successor function. In previous work, we provided an I/O efficient LTL model checking algorithm that builds on topological sort [4]. The algorithm does not work on-the-fly, however, which limits its applicability. In addition, the algorithm is not revisiting resistant. The main contribution of this paper is to overcome these obstacles by providing a new algorithm. The algorithm adapts the idea of the on-the-fly MAP algorithm [9], which is revisiting resistant. In particular, we exploit the algorithm's property of decomposing a graph into several independently processable, smaller sub-graphs. This, in combination with revisiting resistance, significantly improves its practical behavior. We consider several heuristics that guide the decomposition.

Related work. Regarding I/O efficient LTL model-checking, we explicitly compare our work to all existing approaches in Sections 5 and 6. Works on improving the efficiency of delayed duplicate detection (DDD) include hash-based DDD [10], structured DDD [11], graph compression, lossy hash tables and bit-state hashing [12]. All these techniques are orthogonal to our approach and can be combined with the revisiting resistance principle. We have not implemented these other techniques to provide an empirical evaluation.

Main Results. The contribution of this paper can be summarized as follows:

- We explore the notion of a revisiting resistant algorithm and show that the I/O efficient algorithm from [4] is not revisiting resistant (Section 2).
- We present a revisiting resistant I/O efficient reachability algorithm (Section 3).
- We describe the I/O efficient MAP algorithm for LTL model-checking that works *on-the-fly* (Section 4), analyze its theoretical complexity (Section 5), and compare it to other algorithms, both in terms of asymptotic complexity (Section 5) and experimental behavior (Section 6).

2 Revisiting Resistance

In this section, we explain that some algorithms exhibit a quite distinct property that can be of use when adapting the algorithm to an I/O efficient setting. We will refer to this property as *revisiting resistance* and will brand algorithms satisfying the property as *revisiting resistant algorithms*.

We start with a brief description of a general graph search algorithm. Basically, a search algorithm maintains two data structures: a set of vertices that await processing (*open set*) and a set of vertices that have been processed already (*closed set*). The way in which vertices are manipulated by a general algorithm is depicted in Fig. 1(a). A vertex from the open set is selected and its immediate successors are generated (by traversing edges originating from the vertex). The newly generated vertices are checked against the closed set, to ensure that information stored in the closed set is properly updated. Also, if there is need for further processing of some vertices, they are inserted into the open set along with all necessary information for the processing.

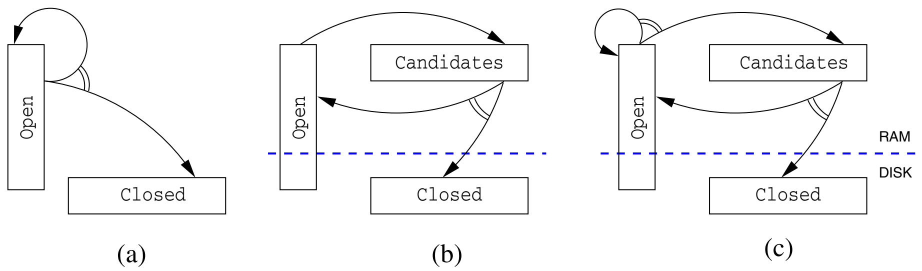
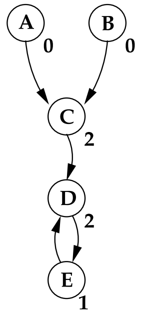


Fig. 1. Vertex work flow: (a) Standard search algorithm, (b) I/O search algorithm with delayed duplicate detection, (c) I/O search algorithm with delayed duplicate detection and revisiting

An I/O efficient search algorithm utilizing the delayed duplicate detection technique has a different vertex work flow. A vertex is picked from the open set and its successors are generated. Then they are inserted into the set of *candidates*, i. e., vertices for which the corresponding check against the closed set has been postponed. In our approach, the set of candidates is kept completely in memory. Candidates are flushed to disk using a *merge* operation under two different circumstances: Either the open set runs empty and the algorithm has to perform a merge to get new vertices into it, or the candidate set is too large and cannot be kept in memory anymore. The merge operation performs the duplicate check of candidate vertices against closed vertices, and inserts those vertices that require further processing into the open set. A schema of the vertex work flow is depicted in Fig. 1(b).

As explained, the merge operation is performed every time the algorithm empties the set of open vertices. Under the standard I/O efficient approach to BFS graph exploration this happens at least after every BFS level. We have observed that for many particular runs of the I/O efficient BFS algorithm, the fact that the merge operation appears after every BFS level is actually a weak point in the practical performance of the algorithm. This is because often a single BFS level contains a relatively small number of vertices,



Open set	Delayed updates to the closed set	Impact of merge operation on the closed set
A,B	$\emptyset$	$(A,0),(B,0),(C,2),(D,2),(E,1)$
B,C	$(C,-1)$ due to $A \to C$	$(A,0),(B,0),(C,1),(D,2),(E,1)$
C,C	$(C,-1)$ due to $B \to C$	$(A,0),(B,0),(C,0),(D,2),(E,1)$
C,D	$(D,-1)$ due to $C \to D$	$(A,0),(B,0),(C,0),(D,1),(E,1)$
D,D	$(D,-1)$ due to $C \to D$	$(A,0),(B,0),(C,0),\boxed{\mathbf{(D,0)}},(E,1)$

Fig. 2. Example computation of the topological-sort based cycle detection algorithm. Values associated with a vertex correspond to the number of immediate predecessors that have not been processed yet. After the computation, vertices that are associated with a zero value lie outside cycles. The algorithm is not revisiting resistant, as vertex D is labeled with a zero value after the merge operation, although it does lie on a cycle.

in comparison to the full graph. Processing them means that the merge operation has to traverse a large disk file, which is costly.

To fight this inefficiency, we suggest a modification in the vertex work flow of an I/O efficient algorithm, as depicted in Fig. 1(c). A vertex, when generated, is inserted not only into the set of candidates, but also into the open set. This causes some of the vertices stored in the candidate set to be revisited. I.e., the "visit" procedure is performed repeatedly for a vertex *without* properly updating its associated information in the closed set residing in external memory. Consequently, some graph algorithms may exhibit incorrect behavior. Revisiting resistant external memory graph algorithms are those, whose correctness is not vulnerable to repeated edge exploration from vertices in the open set. Below, we demonstrate that some algorithms are vulnerable to the revisiting of candidates and become incorrect, while others cope with the revisiting without problems.

We exemplify the concept of revisiting resistance on the *single source shortest path* (SSSP) and the *topological-search based cycle detection* (OWCTY) [13,14,15] algorithms. As for the SSSP algorithm, the procedure that is bound to edge exploration computes a new value for the target vertex, or updates it if the newly computed value is better (lower in this case) than the value stored before. For example: suppose, an edge (u,v) is labeled with the value t, and vertex u is stored in the open set with an associated value $d(u)$, representing the length of the currently shortest known path to vertex u from the source vertex. The procedure computes a new value for vertex v using the formula $d(v) = d(u) + t$. The new pair $(v, d(v))$ is stored in the candidate set and awaits merging with the value stored in the closed set. After the merge, the value stored in the closed set corresponds to the minimum of the originally stored value and the newly computed value. Note, that the resulting value in the closed set is independent from the number of re-explorations of edge (u,v), and, in other words, the number of merges. Even if performed several times, the computation of the minimum among several possible values remains correct. Therefore, we consider the SSSP algorithm as revisiting resistant.

The situation is quite different in the case of the I/O-efficient OWCTY algorithm. The algorithm performs a cycle detection that is based on the recursive elimination of vertices with zero predecessors. At first, the algorithm computes the number of immediate predecessors for every reachable vertex, and then eliminates vertices whose predecessor count drops to zero. During vertex elimination, the predecessor count is decreased for all immediate successors of the eliminated vertex. Thus, when visiting a vertex v from vertex u, the predecessor count stored at vertex v has to be decreased by one. Unfortunately, the value stored for vertex v can be maintained correctly only by costly access to external memory. One way around this dilemma is to only store a delta alongside v in the candidate set. E. g., a pair $(v,-1)$ indicates that there is a new eliminated predecessor of v, and so the value associated with v should be decreased by one. If we now allow the algorithm to further explore vertices below v, it may happen that the edge (u,v) is re-explored again and another pair $(v,-1)$ is inserted into the set of candidates. However, when the set of candidates is merged with the closed set on disk, the predecessor count of vertex v gets decreased *twice*. This violates the correctness of the algorithm. The problem is exemplified in Fig. 2. Therefore, the algorithm is not revisiting resistant.

3 Revisiting Resistant Reachability

This section explains a simple revisiting resistant algorithm, an I/O efficient breadth-first search (Alg. REACHABILITY). The algorithm's sole task is to traverse all vertices, thus we only have to remember for each vertex whether it has been visited or not. Clearly, once a vertex is marked as visited, additional visits do not change this property.

We introduce revisiting resistance to the standard I/O efficient breadth-first search (BFS) procedure as follows. After a single BFS level is fully generated, a decision is made whether the set of candidates will be merged with the closed set, or whether another BFS level will be processed without prior merging. The pseudo-code of function OPENISNOTEMPTY makes this more precise.

According to our observations, omitting the merge operation as long as there is still some unused memory left is not the optimal strategy. Merging with a small closed set might be cheaper than repeatedly re-exploring vertices of the candidate set. We avoid postponing merge operations on small closed sets by introducing a decision formula that builds upon the estimated time needed to fully generate the next BFS level $(n+1)$, and the estimated time needed to perform the merge operation. These estimations are denoted as $estim(t^{gen}_{n+1})$ and $estim(t^{merge}_{n+1})$, respectively, and they are computed from the sizes of open and closed sets as follows:

$$estim(t^{gen}_{n+1}) = t^{gen}_{n} \cdot \frac{|Open_{n+1}|}{|Open_n|} \quad \text{and} \quad estim(t^{merge}_{n+1}) = t^{merge}_{n} \cdot \frac{|Closed_n| + |Open_n|}{|Closed_n|},$$

where $|Open_{n+1}|$ refers to the number of newly discovered vertices w.r.t. *Candidates*. The decision formula is then a simple comparison of the estimated values:

$$estim(t^{gen}_{n+1}) < estim(t^{merge}_{n+1})$$

Finally, note that in our approach the entire set of candidates is kept in memory. However, there is a different approach to I/O efficient reachability, in which both, the

Algorithm 1. REACHABILITY

```
1: $OmittedMergeCount \leftarrow 0$
2: $Candidates \leftarrow \emptyset$
3: $s \leftarrow$ GETINITIALVERTEX()
4: $Closed \leftarrow \{s\}$
5: $Open.push(s)$
6: while OPENISNOTEMPTY() do
7:     $s \leftarrow Open.pop()$
8:     for all $t \in$ GETSUCCESSORS($s$) do
9:         if $t \notin Candidates$ then
10:            $Candidates \leftarrow Candidates \cup \{t\}$
11:            $LastLevel \leftarrow LastLevel \cup t$
12:            if MEMORYISFULL() then
13:                MERGE()
```

Algorithm 2. OPENISNOTEMPTY

```
1: if $Open.isEmpty()$ then
2:     if ESTIMGEN() $<$ ESTIMMERGE() then
3:         $Open.swap(LastLevel)$
4:         $OmittedMergeCount \leftarrow OmittedMergeCount + 1$
5:     else
6:         MERGE()
7: return $\neg Open.isEmpty()$
```

Algorithm 3. MERGE

```
1: for all $s \in Closed$ do
2:     if $s \in Candidates$ then
3:         $Candidates \leftarrow Candidates \setminus \{s\}$
4: if $OmittedMergeCount > 0$ then
5:     $Open \leftarrow (Open \cup LastLevel) \setminus Closed$
6:     $OmittedMergeCount \leftarrow 0$
7: for all $s \in Candidates$ do
8:     if $OmittedMergeCount = 0$ then
9:         $Open.push(s)$
10:    $Closed \leftarrow Closed \cup \{s\}$
11: $LastLevel.clear()$
12: $Candidates \leftarrow \emptyset$
```

candidate and the closed set are stored in external memory [3]. It is possible to combine revisiting resistant algorithms with that approach by employing a similar decision formula to trigger the merge operation.

4 I/O Efficient MAP Algorithm

In this section we design a new revisiting resistant, I/O efficient algorithm for detecting reachable accepting cycles in an implicitly given, directed graph. The algorithm is derived from the *Maximal Accepting Predecessors* (MAP) algorithm [9,16]. We discuss advantages and disadvantages of the algorithm in comparison to other I/O efficient LTL model checking algorithms.

The main idea behind the MAP algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own accepting predecessor. Instead of expensive computing and storing of all accepting predecessors for each (accepting) vertex, the algorithm computes and stores a single representative accepting predecessor for each vertex, namely the maximal one in a suitable ordering of vertices.

Let $G = (V, E, s_0, F)$ be a directed graph, where V is a set of vertices, E is a set of edges, s_0 is an initial vertex, and F a set of accepting vertices. For technical reasons we assume s_0 is not accepting. Let $\prec$ be a linear order on vertices with minimal element $\bot$. Let $u \leadsto^+ v$ denote that there is a directed path from u to v. Then the *maximal accepting predecessor* function map_G is defined as:

$$map_G(v) = \max\left(\{u \in F \mid u \leadsto^+ v\} \cup \{\bot\}\right)$$

Accepting cycle detection is based on the fact that if $v = map_G(v)$, then v lies on an accepting cycle. While the condition is sufficient, it is not necessary—it is possible that $v \neq map_G(v)$ but v lies on an accepting cycle. Moreover, if v is accepting and $map_G(v) \prec v$ then v does not lie on an accepting cycle. Therefore, the MAP algorithm repeatedly removes vertices for which $map_G(v) \prec v$ and recomputes map_G for all vertices.

There is another feature of the MAP algorithm that can be exploited in designing its I/O efficient version. If $map_G(u) \neq map_G(v)$, then u and v cannot lie on the same accepting cycle. This property allows to decompose the graph G into disjoint subgraphs each time map_G values are computed. Let $P(u) = \{v \mid map_G(v) = u\}$. The vertex u is called a *seed* of a *partition* $P(u)$. For any two vertices u, v we have that either $P(u) = P(v)$ or $P(u) \cap P(v) = \emptyset$. The subgraphs are given by disjoint partitions and it is enough to search for accepting cycles in each partition separately. The algorithm thus maintains a queue of pairs $(seed, partition)$, which is initialized with the partition containing all vertices of G and the initial vertex s_0 as its seed. On each partition P the algorithm computes map_G values. If there is a vertex u such that $map_G(u) \in P$, an accepting cycle is detected, otherwise P is split into smaller partitions which are stored for further processing. Note that sub-partitions with $map_G = \bot$ are dropped immediately.

The algorithm obtains the necessary linear ordering on vertices by assigning a unique number to every vertex. This also allows us to store for each vertex the *order value* of its maximal accepting predecessor rather than the maximal accepting predecessor itself.

The basic structure of our new algorithm follows the original MAP algorithm. We maintain a queue *Partitions* of unprocessed partitions as produced by computing map_G. For each partition the algorithm also records its size. If a partition fits into the main memory, we call a standard in-memory algorithm searching accepting cycles, for example, *nested depth-first search* [17].

Procedure MAP propagates the highest order values across the graph in order to compute map_G values for all vertices in a given partition. In essence, the procedure is very similar to external BFS, but it allows to enqueue already explored vertices if it increases their map_G value. We return to this point in the explanation of procedure UPDATEMAP.

Algorithm 4. DETECTACCEPTINGCYCLE(G)

```
Require: $G = (V, E, s_0, F)$
1: $Partitions.push(s_0, V)$
2: while $Partitions \neq \emptyset$ do
3:     $(s, Partition) \leftarrow Partitions.pop()$
4:     if $\|Partition\| > MEMORY\_CAPACITY$ then
5:         MAP$(s, Partition)$
6:     else
7:         NESTED-DFS$(s, Partition)$
8:     FINDPARTITIONS$(Partition)$
9: return No accepting cycle!
```

Algorithm 5. MAP$(seed, Partition)$

```
1: $Open.push(\langle seed, \perp \rangle)$
2: $Closed \leftarrow \{seed\}$
3: $Closed.setMap(seed, \perp)$
4: while OPENISNOTEMPTY() do
5:     $(s, propagate) \leftarrow Open.pop()$
6:     for all $t \in$ GETSUCCESSORS$(s)$ do
7:         STOREORCOMBINE$(t, propagate)$
```

Algorithm 6. STOREORCOMBINE(s, map)

```
1: if $s \in Candidates$ then
2:     $map' \leftarrow Candidates.getMap(s)$
3:     $Candidates.setMap(s,$ MAX$(map, map'))$
4: else
5:     if MEMORYISFULL() then
6:         MERGE()
7:     $Candidates \leftarrow Candidates \cup \{s\}$
8:     $Candidates.setMap(s, map)$
```

In procedure STOREORCOMBINE, we ensure that the currently highest known accepting predecessor for vertex *s* is stored: if some accepting predecessor for *s* has been encountered already, we compare it to *map* and store the higher one. Otherwise, we store *s* and *map*, possibly first making enough memory available through the MERGE operation.

The MERGE procedure joins data stored in internal memory with an external repository. To identify which vertices are new or having their maximal accepting predecessor updated, the procedure traverses all vertices stored on disk and checks whether they are present in the candidate set. For each vertex found in internal memory, MERGE also compares whether the newly found accepting predecessor is higher than the one stored on disk.

Finally, new vertices and vertices with updated accepting predecessor are appended to the open set. New vertices are added to the *Closed* repository, too.

Algorithm 7. MERGE

```
1: $Candidates \leftarrow Candidates \cap Partition$ {Intersection is made trivially by a single traversal across $Partition$}
2: $Updated = \emptyset$
3: for all $s \in Closed$ do
4:     if $s \in Candidates$ then
5:         UPDATEMAP($s$)
6: for all $s \in Candidates$ do
7:     $Open.push(\langle s, Candidates.getMap(s) \rangle)$
8: $New \leftarrow Candidates \setminus Updated$
9: $Closed \leftarrow Closed \cup New$
10: $Candidates \leftarrow \emptyset$
```

Procedure UPDATEMAP is called from MERGE to compare accepting predecessors of a given vertex s, this time taking *all* known information into account. We compare the (so far) highest accepting predecessor for s stored with the candidate set (in memory), the closed set (on disk), and s itself if it is an accepting vertex. Out of those, the maximal vertex (w.r.1t. $\prec$) is stored as new accepting predecessor for s.

We discard s from memory if its accepting predecessor stored with the candidate set is not higher than the one stored with the closed set, as it does not yield any useful information.

After the loop between lines 3–5 in MERGE finishes, the candidate set contains only new vertices and vertices whose accepting predecessor in memory has been greater than the one stored on disk. In addition, we return the set of vertices whose accepting predecessor has been changed.

Algorithm 8. UPDATEMAP(s)

```
1: $map \leftarrow Candidates.getMap(s)$
2: $map' \leftarrow Closed.getMap(s)$
3: if ISACCEPTING($s$) then
4:     $map' \leftarrow \text{MAX}(map', s)$
5: $Closed.setMap(s, \text{MAX}(map, map'))$
6: $Candidates.setMap(s, \text{MAX}(map, map'))$
7: if $map \succ map'$ then
8:     if $s = map$ then
9:         exit Accepting cycle found!
10:    $Updated \leftarrow Updated \cup \{s\}$
11: else
12:    $Candidates \leftarrow Candidates \setminus \{s\}$
13: return $Updated$
```

Procedure FINDPARTITIONS is called from DETECTACCEPTINGCYCLE to identify new sub-partitions in a given partition. Therefore, the procedure sorts vertices in the partition by their map_G values. After that, it only traverses the sorted list of vertices sequentially (loop 4–13) to find the beginning and end of partitions, and also to find the maximal accepting predecessor in the given partition, which is from this moment on regarded to be non-accepting (see line 3 of MAP). Note that the condition on line 6 leaves out a partition with map_G value set to $\bot$, since it does not contain any accepting vertex and thus cannot contain an accepting cycle.

Algorithm 9. FINDPARTITIONS(*Partition*)

```
1:  $Partition.sortByMap()$
2:  $newPartition \leftarrow \emptyset$
3:  $lastMap \leftarrow \bot$ {$\bot$ is the lowest possible value }
4:  for all $(s, map, order) \in Partition$ do
5:      if $lastMap \neq map$ then
6:          if $lastMap \neq \bot$ then
7:              $Partitions.push(seed, newPartition)$
8:          $newPartition \leftarrow \emptyset$
9:      else
10:         $newPartition \leftarrow newPartition \cup \{s\}$
11:     if $map = order$ then
12:         $seed \leftarrow s$
13:     $lastMap \leftarrow map$
14: $Partitions.push(seed, newPartition)$ {Adding last partition}
```

MAP is a revisiting resistant algorithm because it simply traverses the state space and updates map_G values, which are computed as maximum of the values propagated to it. The order and repetition of vertices does not matter, as the maximum stays the same. Henceforward we will refer to the revisiting resistant version of the I/O efficient MAP algorithm as MAP-rr.

Changes in the algorithm are analogous to the modification of the reachability algorithm presented in Sec. 3, but we have to take care of accepting vertices in a special way: between lines 11 and 12 of function UPDATEMAP we put another line:

$$\textbf{if } map' = \text{ORDERNUMBER}(s) \textbf{ then } Updated \leftarrow Updated \cup \{s\}$$

5 Complexity Analysis and Comparison

A widely accepted model for the complexity analysis of I/O algorithms is the model of Aggarwal and Vitter [1], in which the complexity of an I/O algorithm is measured solely in terms of the numbers of external I/O operations. This is motivated by the fact that a single I/O operation is approximately six orders of magnitude slower than a computation step performed in main memory [18]. Therefore, an algorithm that does not perform the optimal amount of work but has lower I/O complexity may be faster in practice, when compared to an algorithm that performs the optimal amount of work, but has higher I/O complexity. The complexity of an I/O algorithm in the model of Aggarwal and Vitter is further parametrized by M, B, and D, where M denotes the number of items that fits into the internal memory, B denotes the number of items that can be transferred in a single I/O operation, and D denotes the number of blocks that can

Table 1. I/O complexity of algorithms for both modes of candidate set storage. Parameter p_{max} denotes the longest path in the graph going through trivial strongly connected components (without self-loops), l_{SCC} denotes the length of the longest path in the SCC graph, h_{BFS} denotes the height of its BFS tree, and d denotes the diameter of the graph.

Algorithm	Worst-case I/O Complexity
Candidate set in main memory	
EJ'	$O((l+\|F\|\cdot\|E\|/M)\cdot scan(\|F\|\cdot\|V\|))$
OWCTY	$O(l_{SCC}\cdot(h_{BFS}+\|p_{max}\|+\|E\|/M)\cdot scan(\|V\|))$
MAP	$O(\|F\|\cdot((d+\|E\|/M+\|F\|)\cdot scan(\|V\|)+sort(\|V\|)))$
Candidate set in external memory	
EJ	$O(l\cdot scan(\|F\|\cdot\|V\|)+sort(\|F\|\cdot\|E\|))$
OWCTY'	$O(l_{SCC}\cdot((h_{BFS}+\|p_{max}\|)\cdot scan(\|V\|)+sort(\|E\|)))$
MAP'	$O(\|F\|\cdot((d+\|F\|)\cdot scan(\|V\|)+sort(\|F\|\cdot\|E\|)))$

be transferred in parallel, i.e., the number of independent parallel disks available. The abbreviations $sort(n)$ and $scan(n)$ stand for $\Theta(N/(DB)\log_{M/B}(N/B))$ and $\Theta(N/(DB))$, respectively. In this section we give the I/O complexity of our algorithm and compare it with the complexity of the algorithm by Edelkamp and Jabbar [3].

Theorem 1. *The I/O complexity of algorithm* DETECTACCEPTINGCYCLE *is*

$$O(|F|\cdot((d+|E|/M+|F|)\cdot scan(|V|)+sort(|V|)))$$

where d is the diameter of a given graph.

Proof. Since each partition is identified by its maximal accepting vertex, at most $|F|$ partitions can be found during traversal. Thus, lines 2–8 in DETECTACCEPTINGCYCLE are repeated at most $|F|$ times, and consequently, procedures MAP and FINDPARTITIONS are called at most $|F|$ times as well. Each call of FINDPARTITIONS costs at most $O(scan(|V|)+sort(|V|))$, because of the dominating sort operation on line 1 and the linear scan in loop 4–13. The I/O complexity of MERGE is $O(scan(|V|))$, because it is dominated by the scan operation across the closed set (loop 3–5) and writing of new and updated vertices to the open set.

There are two main sources of I/O operations in procedure MAP: merge operations and open set manipulation. MERGE is indirectly called on lines 4 and 7. It is called whenever the memory becomes full (at most $|E|/M$ times) or the open set becomes empty (at most d times). Reading of *Open* on line 5 costs at most $O(scan(|F|\cdot|V|)) = O(|F|scan(|V|))$, because each vertex can appear in the open set as many times as its associated accepting predecessor changes. □

For the purpose of comparison, we denote our new algorithm as MAP, the algorithm proposed in [4] as OWCTY and the algorithm of Edelkamp and Jabbar [3] as EJ. MAP and OWCTY store the candidate set internally, while EJ stores it externally by default. In the case the candidate set is sorted externally, it is possible to perform the merge operation on a BFS level independently of the size of the main memory. This approach

Table 2. Partitions after the first iteration of MAP algorithm. Maximums are taken over partitions with some accepting vertex in them.

Experiment	Graph Size	Number of Partitions	Max. Partition Size		Vertices with $map_G = \perp$	
Lamport(5),P4	74,413,141	838,452	454,073	$< 1\%$	38,717,846	52%
MCS(5),P4	119,663,657	3,373,145	108,092	$< 1\%$	60,556,519	51%
Peterson(5),P4	284,942,015	11,451	12,029,114	4%	142,471,098	50%
Phils(16,1),P3	61,230,206	336,339	129,023	$< 1\%$	43,046,721	70%
Rether(16,8,4),P2	31,087,573	33,353	5	$< 1\%$	30,920,813	99%
Szymanski(5),P4	419,183,762	20,064	131,441,308	31%	209,596,444	50%

is suitable for those cases where memory is small, or the graph is orders of magnitude larger. A disadvantage of the approach is the need to sort during each merge operation. Furthermore, it cannot be combined with heuristics, such as Bloom filters and a lossy hash table [12]. Fortunately, all three algorithms are modular enough to be able to work in both modes. Tab. 1 shows the I/O complexities for all three algorithms in both variants.

It can be seen that the upper bound for the complexity of MAP is worse than the one of EJ and OWCTY (in both modes of the candidate set). Nevertheless, we claim that the complexity is reasonable in most cases, for a number of reasons. First, the algorithm usually performs at most two iterations of the loop between lines 2–8 in procedure DETECTACCEPTINGCYCLE: if an accepting cycle was not found during the first iteration, the state space is partitioned into many partitions (as shown in Tab. 2). Furthermore, if the state space was partitioned evenly, then 1000 partitions would be enough to divide a 1 Terabyte state space into blocks sufficiently small to fit into very modestly sized internal memory, by today's standards. Even if some partitions would not fit into main memory yet, another partitioning round usually decreases the maximal partition size enough such that all remaining partitions fit into internal memory. Therefore, it is reasonable to expect that the algorithm becomes fully internal after a very small number of iterations.

Second, d is commonly not proportional to the size of the state space and is usually not much higher than h_{BFS}.

Third, the upper bound $|F| \cdot |V|$ on the number of vertices revisited due to updates of a map_G value is quite coarse. We have measured the amount of map_G updates for the MAP algorithm with a reverse-BFS ordering of vertices. We found that *map* updates take commonly not more than 20% of the graph exploration (see Tab. 4).

Taking this into account, the complexity of MAP could be very close to

$$O((h_{BFS} + |E|/M) \cdot scan(|V|) + sort(|V|))$$

in most practical cases. We note that this equals the complexity of I/O efficient reachability plus sorting the set of vertices. Our measurements confirm this claim, as shown in Tab. 4.

Table 3. Comparison of revisiting techniques and simple I/O efficient reachability

Experiment	Normal (hours)	Revisiting Resistant (hours)	
Lamport(5),P4	02:51:09	01:19:32	46%
MCS(5),P4	03:56:26	02:41:45	68%
Peterson(5),P4	19:38:32	09:02:37	46%
Phils(16,1),P3	02:09:45	01:41:24	77%
Rether(16,8,4),P2	13:54:29	00:29:19	3%
Szymanski(5),P4	51:20:32	17:54:14	34%
On average	100%		46%

Table 4. Run times of reachability and MAP algorithm

Model	Reachability (hours)	MAP (hours)	
Lamport	2:51:09	3:12:09	112%
MCS	3:56:26	4:28:06	113%
Phils	2:09:45	2:29:26	115%

6 Experiments

In order to obtain experimental evidence about the behavior of our algorithm in practice, we implemented an I/O efficient reachability procedure and three I/O efficient LTL model checking algorithms.

All algorithms have been implemented on top of the DiVinE library [19], providing the state space generator, and the STXXL library [20], providing the needed I/O primitives. Experiments were run on 2 GHz Intel Xeon PC, the main memory was limited to 2 GB, the disk space to 60 GB and wall clock time limit was set to 120 hours. Algorithm MAP-rr is a variant of MAP exploiting its revisiting resistance. Algorithm EJ was implemented as a procedure that performs the graph transformation as suggested in [3] and then employs I/O efficient breadth-first search to check for a counter example. Note, that our implementation of [3] does not include the A^* heuristics and hence can be less efficient when searching for an existing counter example. The procedure is referred to as *Liveness as Safety with BFS* (LaS-BFS) [21].

First of all, we have measured the impact of revisiting resistance on procedure Reachability. We have obtained results that demonstrate significant speed-up, as shown in Tab. 3. We have also measured run times and memory consumption of LaS-BFS, OWCTY, MAP and MAP-rr. The experimental results are listed in Tab. 5. We note that just before the unsuccessful termination of LaS-BFS due to exhausting the disk space, the BFS level size still tended to grow. This suggests that the computation would last substantially longer if sufficient disk space would have been available. For the same input graphs, algorithms OWCTY, MAP and MAP-rr manage to perform the verification using a few Gigabytes of disk space only. All the models and their LTL properties are taken from the BEEM project [22].

Measurements on models with valid properties demonstrate that MAP is able to successfully prove their correctness, while LaS-BFS fails. Additionally, MAP's

Table 5. Run times in `hh:mm:ss` format and memory consumption on a single workstation. "OOS" means "out of space".

	LaS-BFS		OWCTY		MAP		MAP-rr	
Experiment	Time	Disk	Time	Disk	Time	Disk	Time	Disk
Valid Properties								
Lamport(5),P4	(OOS)		02:37:17	5.5 GB	03:16:36	5.7 GB	02:37:56	8.5 GB
MCS(5),P4	(OOS)		03:27:05	9.8 GB	04:59:17	10 GB	04:13:21	11 GB
Peterson(5),P4	(OOS)		18:20:03	26 GB	25:09:35	26 GB	15:24:29	27 GB
Phils(16,1),P3	(OOS)		01:49:41	6.2 GB	02:31:33	7.8 GB	02:19:20	8.1 GB
Rether(16,8,4),P2	53:06:44	12 GB	07:22:05	3.2 GB	12:31:18	6.3 GB	00:39:07	6.3 GB
Szymanski(5),P4	(OOS)		45:52:25	38 GB	59:35:25	38 GB	29:09:12	39 GB
Invalid Properties								
Anderson(5),P2	00:00:17	50 MB	07:14:23	3.3 GB	00:00:07	2 MB	00:00:01	4 MB
Bakery(5,5),P3	00:25:59	5.4 GB	68:23:34	38 GB	00:00:09	16 MB	00:00:23	54 MB
Szymanski(4),P2	00:00:50	203 MB	00:20:07	253 MB	00:00:04	2 MB	00:00:02	4 MB
Elevator2(7),P5	00:01:02	130 MB	00:00:25	6 MB	00:00:05	2 MB	00:00:01	3 MB

performance does not differ much from the performance of OWCTY. Moreover, with the use of revisiting resistant techniques, MAP-rr is able to outperform OWCTY in many cases. We observe that specifically in cases with high h_{BFS}—e. g., Rether(16,8,4),P2—time savings are substantial.

A notable weakness of OWCTY is its slowness on models with invalid properties. It does not work on-the-fly, and is consequently outperformed by LaS-BFS in the aforementioned class of inputs. Algorithms MAP and MAP-rr do not share OWCTY's drawbacks, and in fact they outperform both, OWCTY and LaS-BFS on those inputs. This can be attributed to their on-the-fly nature: On all our inputs, a counter example, if existing, is found during the first iteration.

7 Conclusions

We described *revisiting resistance*, a distinct property of graph algorithms, and showed how it can be of practical use to the I/O efficient approach of processing very large graphs. In particular, we described how a simple I/O efficient reachability procedure with delayed duplicate detection can be extended to exploit its revisiting resistance and showed that the extension is valuable in practice. Furthermore, we analyzed existing I/O efficient algorithms for LTL model checking and showed that the OWCTY algorithm is not revisiting resistant. We introduced a new I/O efficient revisiting resistant algorithm for LTL model checking that employs the *Maximal Accepting Predecessor* function to detect accepting cycles. We analyzed the I/O complexity of the new algorithm, and showed that due to the revisiting resistance, the algorithm exhibits competitive runtimes for verification of valid LTL properties while preserving its on-the-fly nature. According to our experimental results, the algorithm outperforms other I/O efficient algorithms on invalid LTL properties even if it is being slowed down with the vertex revisiting.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: Proc. of LICS 1986, pp. 332–344. Computer Society Press (1986)
3. Edelkamp, S., Jabbar, S.: Large-Scale Directed Model Checking LTL. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 1–18. Springer, Heidelberg (2006)
4. Barnat, J., Brim, L., Šimeček, P.: I/O Efficient Accepting Cycle Detection. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 281–293. Springer, Heidelberg (2007)
5. Korf, R.E., Schultze, P.: Large-Scale Parallel Breadth-First Search. In: Proc. of AAAI, pp. 1380–1385. AAAI Press / The MIT Press (2005)
6. Korf, R.E.: Best-First Frontier Search with Delayed Duplicate Detection. In: Proc. of AAAI, pp. 650–657 (2004)
7. Stern, U., Dill, D.L.: Using Magnetic Disk Instead of Main Memory in the Murphi Verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
8. Munagala, K., Ranade, A.: I/O-complexity of graph algorithms. In: Proc. of SODA, Society for Industrial and Applied Mathematics, pp. 687–694 (1999)
9. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
10. Korf, R.E.: Best-First Frontier Search with Delayed Duplicate Detection. In: Proc. of AAAI, pp. 650–657. AAAI Press / The MIT Press (2004)
11. Zhou, R., Hansen, E.A.: Structured Duplicate Detection in External-Memory Graph Search. In: Proc. of AAAI, pp. 683–689 (2004)
12. Hammer, M., Weber, M.: To Store or Not To Store. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2007)
13. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
14. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is There a Best Symbolic Cycle-Detection Algorithm? In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 420–434. Springer, Heidelberg (2001)
15. Ravi, K., Bloem, R., Somenzi, F.: A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 143–160. Springer, Heidelberg (2000)
16. Brim, L., Černá, I., Moravec, P., Šimša, J.: How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In: Proc. of PDMC (2005)
17. Holzmann, G., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: The SPIN Verification System, American Mathematical Society, pp. 23–32 (1996)
18. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv. 33(2), 209–271 (2001)
19. Barnat, J., Brim, L., Černá, I., Šimeček, P.: DiVinE – The Distributed Verification Environment. In: Proc. of PDMC, pp. 89–94 (2005)
20. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard Template Library for XXL Data Sets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)
21. Schuppan, V., Biere, A.: Efficient Reduction of Finite State Model Checking to Reachability Analysis. International Journal on Software Tools for Technology Transfer (STTT) 5(2–3), 185–204 (2004)
22. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)