Semi-external LTL Model Checking*

Stefan Edelkamp[1], Peter Sanders[2], and Pavel Šimeček[3]

[1] Faculty of Informatics, Dortmund University of Technology, Germany
[2] Faculty of Informatics, University of Karlsruhe, Germany
[3] Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract. In this paper we establish c-bit semi-external graph algorithms, – i.e., algorithms which need only a constant number c of bits per vertex in the internal memory. In this setting, we obtain new trade-offs between time and space for I/O efficient LTL model checking. First, we design a c-bit semi-external algorithm for depth-first search. To achieve a low internal memory consumption, we construct a RAM-efficient perfect hash function from the vertex set stored on disk. We give a similar algorithm for double depth-first search, which checks for presence of accepting cycles and thus solves the LTL model checking problem. The I/O complexity of the search itself is proportional to the time for scanning the search space. For on-the-fly model checking we apply iterative-deepening strategy known from bounded model checking.

1 Introduction

Graph search algorithms such as breadth-first search (BFS), depth-first search (DFS), A*, and their variants, play an important role in model checking, as well as in other branches of computer science. All use duplicate detection in order to recognize when the same vertex is reached via alternative paths in a graph. This traditionally involves storing already explored vertices in random access memory (RAM) and checking newly generated vertices against the stored vertices. However, the available amount of RAM severely limits the range of problems that can be solved with this approach. Although many clever memory saving techniques, such as state space reduction, abstraction, and compression, have been developed, all are eventually limited in terms of scalability, and many practical graph search problems are too large to be solved using any of these techniques. Relying on the virtual memory slows down the exploration due to an excessive number of page faults. Over the past few years, several researchers have shown that the scalability of graph search algorithms can be dramatically improved by using external memory, such as disk, to store generated vertices for use in duplicate detection. However, this requires different search strategies to eliminate the impact of the several orders of magnitude difference in random access speed between RAM and disk.

External memory algorithms [20] carefully organize the access to disk. The efficiency of algorithms is then measured in number of block I/O operations performed. The frequently used I/O pattern is external file scanning, processing a stream of records

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338, the Academy of Sciences grant No. 1ET408050503, DFG grant SA 933/3-1 and DFG grant ED 74/4-1.

stored consecutively on disk. If the block size is B, the number of block accesses (I/Os) for scanning N nodes is $O(scan(N)) = O(N/B)$. Another important operation is external sorting. Given that the RAM can contain M nodes it has a complexity of $O(sort(N)) = O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os.

Enumerative model checking is a search in an implicitly given state space graph (*implicit graph*) $G = (V, E)$ induced by an initial vertex s and a successor generation function *succ*. Vertices correspond to states and transitions to edges. The maximal size of the vertex, v_{max}, depends on the encoding and denotes the length of a state vector. *Reachability* is one of the simplest model checking problems. If G is undirected, the complexity of the external memory variant of BFS for solving the reachability problem is bounded by $O(sort(|E|) + scan(|V|))$ I/Os [21]. For directed graphs, the complexity raises to $O(sort(|E|) + l \cdot scan(|V|))$ [26], where $l := \max\{\delta(s,u) - \delta(s,v) + 1 \mid (u,v) \in E\}$ is the length of the longest back-edge in the BFS graph or its *locality* [27]. The locality is bounded by the length of the shortest counter-example $\delta^* := \min\{\delta(s,f) \mid f \in F\}$ (in case of an error) and by the eccentricity of s (max. BFS level) $\epsilon_s := \max\{\delta(s,v) \mid v \in V\}$ (in case of no error), where $F \subseteq V$ is a set of accepting vertices.

Semi-external graph algorithms [1] are algorithms, which allocate $O(|V|)$ *machine words* in the internal memory. Thus, they can store $O(v_{max}) \geq O(\log |V|)$ bits of data per vertex. Since the internal memory is the limiting factor for such algorithms, it makes sense to further reduce the memory requirement to a small constant number of *bits* per vertex. Therefore, we define *c-bit semi-external search algorithms*, which take at most c bits per vertex in the internal memory. Considering small c, with Gigabytes of RAM, and given that state vectors in model checking are large, such algorithms allow us to handle spaces that are orders of magnitudes larger than the main memory.

We present a semi-external solution to the LTL model checking problem, which amounts to finding an accepting cycle in the state space graph of a Büchi automaton [12]. The algorithm we present has an I/O complexity equal to the complexity of I/O-efficient reachability. The approach relies on the I/O-efficient external construction of a *minimal perfect hash function* (MPHF) [7,8], i.e., a one-to-one correspondence between V and $\{0, \ldots, |V| - 1\}$. It allows compressing V to $c|V|$ bits for small constants c. Thereby, we solve a problem considered in a series of preceding papers [15,4,5] that have I/O complexities higher than the reachability analysis.

The paper is organized as follows. First, we define c-bit semi-external algorithms as needed for implicit graph search and explain the generation of space-efficient minimal perfect hash functions. Subsequently, we illustrate our semi-external solution to the LTL model checking problem using the constructed perfect hash function, and analyze its I/O complexity. Afterwards, we address the problem to find the counter-example on-the-fly. For the purpose of comparison, we give a brief overview of related work and put our complexity results into context. And finally, we provide an experimental comparison of existing algorithms for I/O efficient LTL model checking.

2 *c*-Bit Semi-external Graph Algorithms

Semi-external graph algorithms [1] allow to store $O(|V|)$ vertices in the internal memory, thus restricting graph algorithms to store internally only information about vertices, but not about all edges. Although this definition of semi-external graph algorithms appears practical for explicit graphs, it is too general for algorithms on implicit graphs. E.g., almost every internal memory graph algorithm on implicit graphs would be considered as semi-external, since graph edges are given implicitly and there is no need to store them. However, due to large state vector sizes it is apparent that $O(|V|)$ items may easily exceed the amount of available internal memory. For this reason, we study memory consumption in more detail. To derive exact bounds on internal memory consumption, we give a definition of a subclass of semi-external graph algorithms.

Definition 1. *(c-*bit *Semi-External Algorithm) The graph algorithm $\mathcal{A}$ is called c-*bit semi-external *for $c \in \mathbb{R}^+$, if for each implicit graph $G = (V, E)$ the internal memory requirements of $\mathcal{A}$ are at most $O(v_{max}) + c \cdot |V|$ bits.*

$O(v_{max})$ stands for the internal memory consumed by a program code, auxiliary fixed sized variables, and storage of a constant amount of vertices. The value $c \cdot |V|$ stands for the internal memory consumed by information about vertices. Including v_{max} in the complexity is necessary, since this value differs for different graphs[1] – otherwise, for a bound of $O(1) + c \cdot |V|$ bits, we could always find a graph requiring v_{max} that exceeds the constant in $O(1)$, which would prohibit storing even a constant number of vertices. Including the state vector size in the definition of semi-external algorithm also takes a lower bound of $\log\log u + (\log_2 e)|V| + O(\log |V|)$ bits [16] on the space of an MPHF into account, where u denotes the number of all possible states (including unreachable ones) and e is Euler's number.

In the remainder of this section, we refer to results on memory-efficient construction of MPHFs and give sufficient details on the space and the I/O complexity of used algorithms. We introduce a c-bit semi-external depth-first search with use of MPHF. Its I/O complexity is $O(scan(|V|))$ plus the complexity of the hash function construction. There is no known external algorithm computing the DFS order with such a low I/O complexity on general graphs. Third, we extend this DFS implementation to find accepting cycles in order to solve the LTL model checking problem with the same I/O complexity.

2.1 Memory Efficient Minimal Perfect Hash Function

Perfect hashing is a space efficient way of associating unique identifiers with the elements of a static set $V \subseteq U$. A perfect hash function maps $V \subseteq U$ to unique values in the range $\{0, \ldots, N-1\}$, for some appropriate value of N. A minimal perfect hash function is a perfect hash function with $N = |V|$. Consequently, a minimal perfect hash function is a one-to-one correspondence between V and $\{0, \ldots, |V|-1\}$.

Surprisingly, after 23 years of research, an asymptotically space optimal, practical algorithm for generating MPHFs was recently discovered [7]. The external memory variant, referred to as EPH algorithm, was given in [8]. Although the I/O complexity of this EPH algorithm is not analyzed in [8], it is clear that it is dominated by the need to

[1] The binary vertex representation $v_{\max}$ takes obviously at least $\log |V|$ bits. In the model checking case, it is usually much more.

```
Procedure Perform-DFS$(s, succ)$
  $V :=$ Enumerate-BFS$(s, succ)$
  $h :=$ Construct-MPHF$(V)$
  Depth-First-Search$(s, succ, h)$

Procedure Depth-First-Search$(s, succ, h)$
  Vars: visited : Internal Bit Array$[1..n] = (0,..,0)$
        dfsStack : External Stack of Vertices
  visited$[h(s)] := 1$
  dfsStack.push$(s)$
  while (not dfsStack.empty())
    $u :=$ dfsStack.top()
    if $\exists v \in succ(u).$ visited$[h(v)] = 0$ then
      dfsStack.push$(v)$
      visited$[h(v)] := 1$
    else
      dfsStack.pop()
```

Fig. 1. c-bit Semi-External Depth-First Search

sort all items by their hash signature in a partitioning step. MPHFs constructed by EPH can be stored in less than 4 bits per item.[2]

The Heuristic EPH algorithm, published also in [8], differs from EPH in the choice of the hash function. It results in a substantial speed-up in both construction and search times, but incurs additional memory overhead per bucket. In our implementation, it needs 1 additional bit per vertex, i.e., our implementation of the Heuristic EPH requires 5 bits per vertex.

2.2 Depth-First Search

The main observation for graph search is that given a perfect hash function h, algorithms like plain DFS, BFS, and A* need only one bit per vertex storing whether it has already been visited. The general approach applying a bit-array for tracking reached vertices in DFS is illustrated in Fig 1 as procedure *Depth-First-Search*.

Our algorithm first enumerates all reachable vertices using external BFS, which performs $O(l \cdot scan(|V|) + sort(|E|))$ operations (*Enumerate-BFS*) [18,26]. Then the EPH algorithm constructs the MPHF with I/O complexity $O(sort(|V|))$ (*Construct-MPHF*) – this complexity follows from [8], although it is not explicitly stated there.

The stack in procedure *Depth-First-Search* can be stored on disk. The procedure performs exactly $|V|$ *push* operations and $|V|$ *pop* operations. It is easy to implement the stack in the way that the I/O-complexity of the procedure is $O(scan(|V|))$. Thus the overall I/O complexity of the algorithm (procedure *Perform-DFS*), including graph generation and hash function construction, is $O(l \cdot scan(|V|) + sort(|E|) + sort(|V|) + scan(|V|)) = O(l \cdot scan(|V|) + sort(|E|) + sort(|V|))$. In implicit graphs, we have

[2] Although [8, Table 2] shows a value higher than 4 bits per item for $n = 10^8$, it is caused by a poor choice of the *bucket* size; i.e., $b = 20$ causes buckets to contain less than 96 items per bucket on average. In our implementation of EPH, we always choose a better bucket count, which guarantees that each bucket contains at least 128 items on average. This amount assures that the fixed cost per bucket is divided into sufficiently many items to keep the overall costs below 4 bits per item.

```
Procedure Perform-DDFS($s, succ$)
   Vars: $V$ : Vertex Set;
         $h$: Perfect Hash Function;
   $V :=$ Enumerate-BFS($s, succ$)
   $h :=$ Construct-MPHF($V$)
   Double-Depth-First-Search($s, succ, h$)

Procedure Double-Depth-First-Search($s, succ, h$)
   Vars: visited : Internal Bit Array$[1..n] := (0,..,0)$;
         $F$ : List of Accepting Vertices;
   $F :=$ Depth-First-Search-1($s, succ, h$)
   visited $:= (0, ..., 0)$
   for each $i$ in $F$ do
      if visited$[h(i)] = 0$ then
         if (Depth-First-Search-2($i, succ, h$))
            return 'cycle found'
   return 'no cycle'
```

Fig. 2. c-bit Semi-External Double Depth-First Search

$|V| < |E|$, because all vertices contained in V induced by s and *succ* are reachable. Therefore the I/O complexity simplifies to $O(l \cdot scan(|V|) + sort(|E|))$.

With EPH minimum perfect hashing the algorithm is 5-bit semi-external, since less than 4 bits per vertex are needed for storing h, and 1 bit per vertex is needed for *visited*.

2.3 Double Depth-First Search

The LTL model checking problem amounts to detecting accepting cycles in the global state space graph. It is possible to find an accepting cycle with the *double depth-first search* algorithm [13, Algorithm A]. The algorithm performs the first DFS to find a list F of all accepting vertices sorted in DFS postorder. The second DFS explores the graph gradually from all vertices in F. The pseudo code of this algorithm is shown in Fig. 2.

Depth-First-Search-1 is a modified version of *Depth-First-Search*, which appends an accepting vertex to F, while it is removed from *dfsStack*. *Depth-First-Search-2* is a modified version of *Depth-First-Search*, which finishes with return value *true*, if it wants to add its initial vertex to *dfsStack* in the main loop (and so it finds a path from the initial vertex to itself). For simplicity and memory efficiency, we assume that array *visited* is shared by both procedures. The correctness of the algorithm is proven in [13].

These two modifications of *Depth-First-Search* have clearly the same I/O complexity as the original procedure. Therefore, the overall I/O complexity of *Perform-DDFS* remains at $O(l \cdot scan(|V|) + sort(|E|))$. Moreover, they share the same hash function and memory space for the *visited* field. As in the DFS case, with EPH the algorithm is 5-bit semi-external, since less than 4 bits per vertex are needed for storing h, and 1 bit per vertex is needed for *visited*.

2.4 General Graph Search

We have shown a way to solve the LTL model checking problem using double depth-first search with a visited vertex set represented in form of a minimum perfect hash function. We have chosen double depth-first search, because it is the most time and memory efficient algorithm for searching accepting cycles in the internal memory and it sustains the efficiency in semi-external setting. For example, nested depth-first search [13] has higher memory demands and its on-the-fly nature is not a big advantage, since for MPHF construction, the algorithm would have to enumerate the entire vertex set anyway.

```
Procedure General-Search
    Vars: $V$ : Vertex Set;
          $h$: Perfect Hash Function;
    $V := \mathit{External\text{-}BFS}(s, succ)$
    $h := \mathit{Construct\text{-}MPHF}(V)$
    $\mathit{Search}(s, succ, h)$
```

Fig. 3. General c-bit Semi-External Graph Search

Besides combination of MPHF and LTL model checking, we can also consider other applications in model checking. E.g., for global CTL model checking [12], graph decomposition to strongly connected components (SCCs) is needed, which is easy using Kosaraju-Sharir's algorithm (SCC decomposition using forward and backward DFS [25]). If the implicit definition also contains backward successor generation, the algorithm for SCC decomposition is straightforward. This way, employing MPHF gives us a handle to many space efficient semi-external model checking algorithms for the prize of single state space generation needed for MPHF construction. In fact, all such general search algorithms on implicit graphs can follow the semi-external exploration procedure as outlined in Fig.3.

3 On-the-Fly LTL Model Checking

The idea of an increasing depth bound to obtain short lasso shaped counter-examples as witnesses for a falsified LTL property refers to pioneering work of [6], which searches for a counter-example in the state space graph unrolled to a fixed depth k.

A similar iterative-deepening strategy can be easily applied to our case. Since we use external breadth-first search to generate the state space, we can search for a counter-example every time a new level is generated. This approach has two main advantages

- The counter-example can be found before the entire graph is generated – it is found *on-the-fly*. Since graph generation is the main source of I/Os, performance can be significantly improved on inputs with existing counter-examples.
- It can produce a shorter counter-example, since the depth for its search is limited. However, the counter-example is not necessarily the shortest.

The algorithm is derived from the one in Section 2.3 by unwinding procedure *Enumerate-BFS* and moving MPHF construction and DDFS inside BFS levels generation as shown in Fig. 4.

Every search for a counter-example in an incomplete graph applies $O(sort(|V|))$ I/Os, determined by the I/O complexities of *Construct-MPHF* and *Double-Depth-First-Search*. The search for a counter-example is performed after the generation of each BFS level (one BFS iteration). With $\epsilon_s := \max\{\delta(s, v) \mid v \in V\}$ at most ϵ_s BFS iterations are invoked. Moreover, the generation of every BFS level requires $O(scan(|V|))$ I/Os. Therefore, the overall I/O complexity of the algorithm is

$$O(\epsilon_s \cdot sort(|V|) + l \cdot scan(|V|) + sort(|E|)) = O(\epsilon_s \cdot sort(|V|) + sort(|E|)).$$

```
Procedure Perform-IDDFS$(s, succ)$
  Vars: $V$ : Vertex Set;
        $h$: Perfect Hash Function;
        $nextLevel$: Set of Vertices;
  $nextLevel := \{s\}$
  while $nextLevel \neq \emptyset$ do
    $V := V \cup nextLevel$
    $h := Construct\text{-}MPHF(V)$
    $Double\text{-}Depth\text{-}First\text{-}Search(s, succ, h)$
    nextLevel $:= succ(nextLevel) \setminus V$
```

Fig. 4. On-the-fly Semi-External Double Depth-First Search

Although the I/O complexity of double DFS is only $scan(|V|)$, in practice, due to the efforts for generating the successors of a vertex, the run time of DFS search often substantially exceeds the run time of hash function construction (even though its I/O complexity is $sort(|V|)$). Therefore, the internal memory search for a counter-example is too expensive to be invoked after the generation of each BFS level. For this reason, we implemented the algorithm in such a way that it measures run times of checks for a counter-example (including hash function generation) and tries to predict the run time on the next level. This refined threshold determination algorithm invokes a check for counter-examples, only if the predicted time sufficiently amortizes the time for graph generation.

4 Related Work

Since model checking amounts to graph search, our algorithm is strongly related to external memory graph algorithms [20,11]. Most results consider the graph to be explicitly given in the external memory. Graph algorithms on explicit graphs, however, suffer from the storage of edges in the external memory, which causes one I/O operation each time they need to access successors of a given vertex. It brings at least $|V|$ additional I/Os. The situation has been slightly improved for undirected graphs, where the I/O complexity improves to $O(\sqrt{|V| \cdot scan(|V|+|E|)} + sort(|V|+|E|))$ [19], but in general, state spaces in model checking are directed.

Fortunately, practical model checking is performed on state spaces given by a system model – i.e., implicit graph definition. Thus, it avoids the expensive fetching of edges. However, in contrast to the explicit case, the implicit representation of edges does not allow to store the information about explored edges, which is essential to avoid re-exploration. For this reason, it is not trivial to make algorithms on explicit graphs work also on implicit graphs efficiently. For example, there is no known efficient implementation of depth-first search for implicit graphs.

Therefore, algorithms refer to a breadth-first traversal through the graph and employ the *delayed duplicate detection* technique [18,21,26]. The search procedure has to maintain a set of visited vertices to prevent their re-exploration. Since the graphs are large, the visited set cannot be kept completely in main memory. Most of it is stored on an external memory device. When a new vertex is generated, it is checked against the visited set to avoid its re-exploration. The idea of the delayed duplicate detection technique is to postpone the individual checks and perform them together in a group, for the price of a single scan operation. The group of vertices waiting for checking against the visited set is called *candidate set*. There are two basic kinds of duplicate detection: The one making an internal memory a buffer for candidate set and the one storing candidate set in the external memory. The first has an advantage that no sorting is needed during duplicates removal. The latter is better, when candidate sets are too large to fit in the internal memory and thus, using the first approach they would have to be divided into several pieces and checked separately. Complexities of both approaches are different, and incomparable in general.

4.1 External LTL Model Checking

The first I/O-efficient solution for the LTL model checking problem by Edelkamp and Jabbar [15] builds on the reduction of liveness to the safety approach by Schuppan and Biere [24] designed for symbolic exploration with BDDs. It operates on-the-fly and applies guidance for checking liveness properties [15] with a set of heuristic functions.

Barnat et al. proposed another I/O efficient algorithm [4] for accepting cycle detection. It applies the OWCTY (One Way Catch Them Young) algorithm [23,10] – an accepting cycle detection algorithm based on topological sort. The algorithm itself is an off-line algorithm. It generates the whole state space and then iteratively prunes the parts of the state space that do not lead to any accepting cycle. The underlying exploration strategy is breadth-first based. Later, they also proposed an on-the-fly algorithm [5] based on the MAP (Maximal Accepting Predecessors) algorithm [9].

All three approaches were theoretically compared, experimentally evaluated and each of them has shown its practical applicability to a certain class of problems.

4.2 Complexity Comparison

In this section, we compare the new semi-external approach to the existing external LTL model checking algorithms, in terms of internal memory consumption and I/O complexity (see Table 1).

Regarding the I/O complexity, the new algorithms contributed in this paper compete much better than previous work. The off-line version (DDFS) has the same I/O complexity as the external breadth-first search, which defeats existing algorithms substantially. The on-the-fly variant (*IDDFS*) is worse than off-line, but it is still reasonable compared to the rest of the algorithms. Table 1 shows I/O complexities of all algorithms and also gives I/O complexities of their versions with candidate set stored in RAM. Note that in this case we consider a variant of the EPH algorithm with I/O complexity $O(n/M \cdot scan(n))$ (different bound for external sort) rather than $O(sort(n))$, because it simplifies resulting complexities. The existing external memory algorithms are named after their internal memory variants: L2S [15], OWCTY [4] and MAP [5].

The disadvantage of our semi-external algorithm is that it needs $\Omega(|V|)$ bits in the internal memory – thus we can always find a graph, on which it exceeds an available internal memory. For example, for 5-bit semi-external search on a computer with 2 GB RAM, the algorithm cannot handle graphs with more than $2 \cdot 2^{30} \cdot 8/5 \approx 3 \cdot 10^9$ vertices

Table 1. I/O complexities for LTL model checking

	Candidate Set on Disk	Candidate Set in RAM
L2S	$O(l \cdot scan(f \cdot n) + sort(f \cdot m))$	$O((l + f \cdot m/M) \cdot scan(f \cdot n))$
OWCTY	$O(\tau \cdot ((\epsilon_s + \psi) \cdot scan(n) + sort(m)))$	$O(\tau \cdot (\epsilon_s + \psi + m/M) \cdot scan(n))$
MAP	$O(f \cdot ((d+f) \cdot scan(n) + sort(f \cdot m)))$	$O(f \cdot ((d+m/M+f) \cdot scan(n) + sort(n)))$
DDFS	$O(l \cdot scan(n) + sort(m))$	$O((l + m/M) \cdot scan(n))$
IDDFS	$O(\epsilon_s \cdot sort(n) + sort(m))$	$O((\epsilon_s + m/M) \cdot n/M \cdot scan(n))$

$m = |E|$ ... number of edges, $n = |V|$ ... number of vertices,
$f = |F|$ ... number of accepting vertices, τ ... length of the longest path in the SCC graph,
ϵ_s ... eccentricity of the initial vertex, d ... diameter of the graph,
l ... locality, i.e., length of the longest back edge in breadth-first search graph
ψ ... length of the longest path in the graph going through trivial strongly connected components (without self-loops).

(since it stores 5 bits per vertex internally). In contrast, purely external algorithms are limited only by the capacity of the external memory. Nevertheless, considering that one vertex is v_{max} bytes long, we get that, with 2 Gigabytes of RAM our algorithm can handle state spaces which need approximately $3 \cdot v_{max}$ Gigabytes to be stored externally. For practical values of v_{max} (20-1000 from our experience on models from the Benchmark for Explicit Model Checkers [22]) the state space would be hundreds or thousands of Gigabytes large. When manipulating such a large piece of data, our algorithm takes advantage of lower I/O complexity and as a result, it can be much faster than previous algorithms, which makes a price of 5 bits of the internal memory per vertex quite reasonable.

5 Experimental Results

In order to obtain experimental evidence about the behavior of our algorithm in practice, we implemented three existing external memory LTL model checking algorithms (as introduced in Section 4) and compared their run times and allocated disk space to both versions (DDFS and IDDFS) of the new semi-external algorithm.

All algorithms have been implemented on top of the DiVinE library [3], providing the state space generator, and the STXXL library [14], providing the I/O primitives. Experiments were run on a Linux workstation with 2 GHz Intel Xeon processor, the main memory was limited to 2 GB, the disk space to 60 GB and wall clock time limit was set to 120 hours. For compilation of sources we used GNU C++ compiler with optimization level 2. Algorithm L2S was implemented as a procedure that performs the graph transformation as suggested in [15] and then employs I/O efficient breadth-first search to check for a counter-example. Note, that our implementation of L2S does not include the A* search heuristics and, hence, can be less efficient when searching for an existing counter-example. Algorithms DDFS and IDDFS were implemented using Heuristic EPH [8] (for the sake of speed), thus one additional bit per vertex is allocated in comparison to EPH.

Table 2. Experimental results for different I/O-efficient algorithms

	L2S		OWCTY		MAP		DDFS		IDDFS	
Experiment	Time	Disk	Time	Disk	Time	Disk	Time	Disk	Time	Disk
Valid Properties										
Elev.2(16),P4	(OOS)		09:54	9.2 GB	07:45	16 GB	08:01	10 GB	08:03	10 GB
Lamport(5),P4	(OOS)		02:37	5.5 GB	03:16	5.7 GB	02:15	3.3 GB	02:16	3.3 GB
MCS(5),P4	(OOS)		03:27	9.8 GB	04:59	10 GB	03:42	6.2 GB	03:59	6.2 GB
Peterson(5),P4	(OOS)		18:20	26 GB	25:09	26 GB	14:19	16 GB	18:37	16 GB
Phils(16,1),P3	(OOS)		01:49	6.2 GB	02:31	7.8 GB	02:26	6.7 GB	02:54	6.7 GB
Ret.(16,8,4),P2	53:06	12 GB	07:22	3.2 GB	12:31	6.3 GB	06:26	3.4 GB	07:52	3.4 GB
Szyman.(5),P4	(OOS)		45:52	38 GB	59:35	38 GB	30:36	24 GB	34:21	24 GB
Invalid Properties										
Bakery(5,5),P3	00:25	5.4 GB	68:23	38 GB	<1m	16 MB	36:48	29 GB	00:01	71 MB
Szyman.(4),P2	00:00	203 MB	00:20	253 MB	<1m	2 MB	00:10	237 MB	<1m	8 MB
Elev.2(7),P5	00:01	130 MB	<1m	6 MB	<1m	2 MB	<1m	4 MB	<1m	6 MB
Lifts(7),P4	00:01	59 MB	00:28	475 MB	<1m	4.6 MB	00:32	561 MB	00:07	239 MB

Times are given in `hh:mm` format, "OOS" = "out of space", "<1m" = "below 1 minute".

Table 3. Size of used models and internal memory used for storage of MPHF

Model	Number of Vertices	v_{max}	ϵ_s	MPHF Size (bits/vertex)
Elev.2(16),P4	173,916,122	30 bytes	94	4.941
Lamport(5),P4	74,413,141	24 bytes	99	4.941
MCS(5),P4	119,663,657	28 bytes	91	4.941
Peterson(5),P4	284,942,015	32 bytes	177	4.941
Phils(16,1),P3	61,230,206	50 bytes	47	4.941
Ret.(16,8,4),P2	31,087,573	91 bytes	553	4.941
Szyman.(5),P4	419,183,762	32 bytes	223	4.941

The experimental results are listed in Tab. 2. Names of algorithms correspond to names in Section 4. We note that just before the unsuccessful termination of L2S 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, DDFS and IDDFS managed 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 DDFS is able to successfully prove their correctness, while L2S fails. Additionally, DDFS is faster than OWCTY on most of inputs and outperforms MAP on all inputs except for model $Elev.2(16), P4$. We observe that DDFS is especially better than OWCTY on inputs where the eccentricity of the initial vertex (Tab. 3) is high, since the first enumeration phase costs almost the same time as the initial iteration of OWCTY, but the double depth-first search is not influenced by the eccentricity, while the other iterations of OWCTY are.

A notable weakness of DDFS is its bad performance on models with invalid properties. It does not work on-the-fly, and hence is outperformed by L2S and MAP on some inputs. For this reason, the iterative-deepening variant (IDDFS) has been proposed in

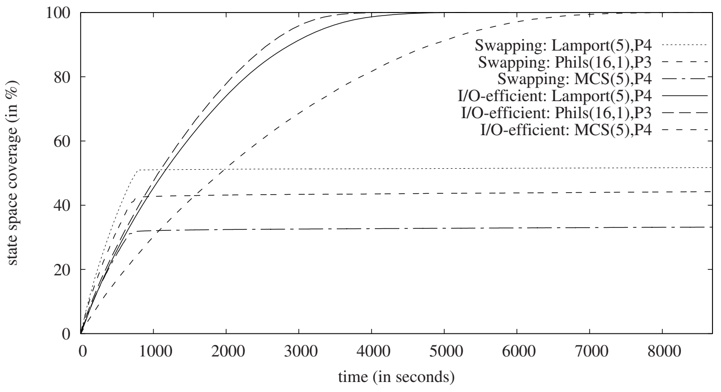

Fig. 5. Comparison of BFS with virtual memory swapping and I/O-efficient BFS

Section 3. It is a little bit slower than DDFS on inputs with valid properties (since all intermediate checks are useless in that case), but our measurements confirm that it is able to find a counter-example much sooner. Actually, its run times are close to run times of MAP, which appears to be the best choice for models with invalid properties. In our implementation, IDDFS is designed to keep costs for intermediate checks below 20% of the run time.

We also measured the internal memory taken for MPHFs representing state spaces (Tab. 3). Measured amounts of bits per vertex confirm theoretically achieved estimations for Heuristic EPH.

Finally, to support a need for I/O-efficient algorithms, we demonstrate in Fig. 5 that after exceeding the main memory, BFS with use of virtual memory swapping almost stops the exploration due to excessive amount of page faults. In contrast, I/O-efficient BFS is able to finish in reasonable time. A similar observation has been made in [2].

6 Conclusion and Discussion

With this paper we contribute c-bit semi-external DFS search for validating safety and c-bit semi-external double DFS for validating liveness properties. For bug-hunting, we implemented an iterative deepening variant of double DFS using the same amount of RAM. With minimum perfect hashing with EPH, we obtained a c-value of about 5, with Heuristic EPH used in the experiments we validated that c is less than 6.

With double DFS (DDFS) we are in many cases faster than all previous algorithms for LTL model checking L2S [15], OWCTY [4], MAP [5] (in theory and practice). Moreover, we saw that solving the LTL model checking problem off-line is not more involved than state space enumeration.

As a drawback, semi-external DDFS is neither optimal, nor on-the-fly. We discussed improvements for transforming DDFS into an on-the-fly algorithm using iterative deepening, but currently we lack an algorithm that is linear wrt. generating the search space. The algorithm by Edelkamp and Jabbar [15] operates on-the-fly, can be directed towards the error, and produces optimal counter-examples, but traverses the crossproduct graph, which can be too large in many cases. MAP has a considerable I/O complexity, and is on-the-fly only for a restricted number of properties. Another open question is the design of optimal counter-examples providing LTL model checking algorithm that is linear in the size of the search space.

A notion of c-bit semi-external algorithms makes a space consumption estimates much closer to the theoretical lower bound [16]. An interesting question is how small we can get the c for c-bit semi-external DFS. Both in theory and practice, we can do somewhat better by using perfect hash functions (PHFs) with range $\{0, \ldots, m-1\}$ for $m = \Theta(n)$ rather than minimal PHFs (with $m = n$). We need to allocate m bits for storing visited-bits, but we need less space for representing the hash function. Botelho et al. [7] cites a theoretical bound of $(1+(m/n-1)\ln(1-n/m))n \log e$ bits for storing the PHF.[3] Adding m and optimizing, we obtain an optimal value of $m \approx 1.302n$ yielding a total space consumption of about $2.108n$ bits. Botelho et al. give a practical scheme based on 3-uniform hypergraphs with $m \approx 1.23n$ that uses about $1.95n$ bits for the PHF so that we need about $3.18n$ bits in total which is only about n bits off the theoretical bound. Even for the faster and simpler construction using 2-uniform hypergraphs, we get a slight improvement over the 5 bit solution than we obtain with MPHFs: using $m \approx 2n$, we need about $2n$ bits for storing the PHF yielding total space about $4n$. Note that this solution is even more computationally efficient than MPHF based schemes, since it saves a compression step needed to construct an MPHF from a PHF.

Because of external duplicate detection, vertex enumeration is time consuming. There are different possible approaches to tackle the problem. Given a sufficient number of file pointers, external sorting can be reduced to at most two scans over the search space. Moreover, pipelining [14] helps a lot in reducing the number of I/Os in BFS. Furthermore, by faster random access time, flash media might additionally reduce the run time [17].

Due to all of these techniques, we believe that external memory model checking is practical and can be made even more time and memory efficient.

References

1. Abello, J., Buchsbaum, A.L., Westbrook, J.: A functional approach to external graph algorithms. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 332–343. Springer, Heidelberg (1998)
2. Barnat, J.: Distributed Memory LTL Model Checking. PhD thesis, Masaryk University Brno, Faculty of Informatics (2004)
3. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification (Tool Paper). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. 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. Barnat, J., Brim, L., Šimeček, P., Weber, M.: Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In: Proc. of TACAS. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)

[3] For simplicity, we drop all lower order terms and arbitrarily small constants appearing in the actual bounds.

6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) ETAPS 1999 and TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
7. Botelho, F.C., Pagh, R., Ziviani, N.: Simple and space-efficient minimal perfect hash functions. In: Dehne, F.K.H.A., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 139–150. Springer, Heidelberg (2007)
8. Botelho, F.C., Ziviani, N.: External perfect hashing for very large key sets. In: Conference on Information and Knowledge Management (CIKM), pp. 653–662. ACM, New York (2007)
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. 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)
11. Chiang, Y.-J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Symposium on Discrete Algorithms (SODA), pp. 139–149. Society for Industrial and Applied Mathematics (1995)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
13. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. 1(2-3), 275–288 (1992)
14. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard template library for XXL data sets. In: Proc. of ESA. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)
15. 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)
16. Fredman, M.L., Komlós, J.: On the size of separating systems and families of perfect hash functions. SIAM Journal on Algebraic and Discrete Methods 5(1), 61–68 (1984)
17. Gal, E., Toledo, S.: Algorithms and data structures for flash memories. ACM Computing Surveys 37(2), 138–163 (2005)
18. Korf, R.E.: Delayed duplicate detection: Extended abstract. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 1539–1541 (2003)
19. Mehlhorn, K., Meyer, U.: External-memory breadth-first search with sublinear I/O. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 723–735. Springer, Heidelberg (2002)
20. Meyer, U.: Algorithms for Memory Hierarchies. Springer, Heidelberg (2003)
21. Munagala, K., Ranade, A.: I/O-complexity of graph algorithms. In: Symposium on Discrete Algorithms (SODA), pp. 687–694. Society for Industrial and Applied Mathematics (1999)
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)
23. 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)
24. Schuppan, V., Biere, A.: Efficient reduction of finite state model checking to reachability analysis. Int. Journal on Software Tools for Technology Transfer 5(2–3), 185–204 (2004)
25. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers and Mathematics with Applications 7(1), 67–72 (1981)
26. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the Murϕ verifier. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 172–183. Springer, Heidelberg (1993)
27. Zhou, R., Hansen, E.: Breadth-first heuristic search. In: Int. Conf. on Automated Planning and Scheduling (ICAPS) pp. 92–100. AAAI Press / The MIT Press (2004)