I/O Efficient Directed Model Checking

Shahid Jabbar and Stefan Edelkamp

Department of Computer Science
Baroper Str. 301
University of Dortmund, Dortmund, Germany
{shahid.jabbar,stefan.edelkamp}@cs.uni-dortmund.de

Abstract. Directed model checking has proved itself to be a useful technique in reducing the state space of the problem graph. But still, its potential is limited by the available memory. This problem can be circumvented by the use of secondary storage devices to store the state space. This paper discusses directed best-first search to enhance error detection capabilities of model checkers like SPIN by using a streamed access to secondary storage. We explain, how to extend SPIN to allow external state access, and how to adapt heuristic search algorithms to ease error detection for this case. We call our derivate IO-HSF-SPIN. In the theoretical part of the paper, we extend the heuristic-based external searching algorithm to general weighted and directed graphs. We conduct experiments with some challenging protocols in Promela syntax like GIOP and dining philosophers and have succeeded in solving some hard instances externally.

1 Introduction

Model checking [3] has evolved into one of the most successful verification techniques. Examples range from mainstream applications such as protocol validation, software and embedded systems' verification to exotic areas such as business work-flow analysis, scheduler synthesis and verification.

There are two primary approaches to model checking. *Symbolic model checking* [2] uses a representation for the state set based on boolean formulae and decision diagrams. Property validation amounts to some form of symbolic fix-point computation. *Explicit-state model checking* uses an explicit representation of the system's global state graph. Property validation amounts to a partial or complete exploration of the state space. The success of model checking lies in its potential for *push-button* automation and in its error reporting capabilities. A model checker performs an automated complete exploration of the state space of a model, usually applying a depth-first search strategy. When a property violating state is encountered, the search stack contains an error trail that leads from an initial system state to the encountered state. This error trail greatly helps engineers in interpreting validation results.

The sheer size of the reachable state space of realistic models imposes tremendous challenges on the algorithm design for model checking technology. Complete exploration of the state space is often impossible, and approximations are needed. Also, the error trails reported by depth-first search model checkers are often exceedingly lengthy – in many cases they consist of multiple thousands of computation steps which greatly hampers error interpretation. The use of *directed model checking* [6] renders erstwhile unanalyzable problems analyzable in many instances. The quality of the results obtained with heuristic search algorithms like A* [9] depends on the quality of the heuristic estimate. Various heuristic estimates have been devised that are specific for the validation of concurrent software, such as specific estimates for reaching deadlock states.

Nonetheless, the search spaces that are generated during the automated verification process are often too large to fit into main memory. One solution studied in this paper is external exploration. In this case, during the algorithm only a part of the graph is processed at a time; the remainder is stored on a disk. The block-wise access to secondary memory has led to a growing attention to the design of *I/O-efficient* algorithms in recent years. Algorithms that explicitly manage the memory hierarchy can lead to substantial speedups, since they are more informed to predict and adjust future memory access.

In this paper, we address explicit model checking on secondary memory. First we recall the most widely used computation model to design and analyze external memory algorithms. This model provides a basis to analyze external memory algorithms by counting the data transfers between different levels of memory hierarchy. Then, we recall *External* A^*, which extends delayed duplicate detection to heuristic search for the case of implicit (graph is generated on the fly), undirected, and unweighted graphs. In Section 4, we extend the External A* algorithm for the case of directed and weighted implicit graphs - a usual case with state space graphs that appear in model checking. Weighted graphs introduce new issues to the problem. One of the main issue is the presence of negative weights in the graph. These issues are also dealt with further in this section along with proofs of optimality for these extensions.

The second part of the paper mainly deals with practical aspects of external model checking. For the implementation of our algorithms, we chose the experimental model checker HSF-SPIN as the basis. HSF-SPIN (Section 5) extends SPIN by incorporating heuristics in the search procedure. It has shown a large performance gain in terms of expanded nodes in several protocols. We call our extension as IO-HSF-SPIN and is discussed in Sections 6. For the experiments, we choose three protocols in Promela syntax: Dining philosophers, Optical Telegraph, and CORBA-GIOP. In Section 7, we illustrate the efficiency of our approach on these protocols, by the sizes of problems that we have succeeded in solving. Finally, we address related and future work and draw conclusions. For the convenience of readers, an appendix is set at the end of the paper that discusses the proofs of some of the key theorems referenced in this paper.

2 I/O Efficient Algorithms

The commonly used model for comparing the performance of external algorithms [18] consists of a single processor, a small internal memory that can hold up to M data items, and an unlimited secondary memory. The size of the input problem (in terms of the number of records) is abbreviated by N. Moreover, the *block size* B governs the bandwidth of memory transfers. It is often convenient to refer to these parameters in terms of blocks, so we define $m = M/B$ and $n = N/B$. It is usually assumed that at the beginning of the algorithm, the input data is stored in contiguous block on external memory, and the same must hold for the output. Only the number of blocks' reads and writes are counted; computations in internal memory do not incur any cost. An extension of the model considers D disks that can be accessed simultaneously. When using disks in parallel, the technique of *disk striping* can be employed to essentially increase the block size by a factor of D. Successive blocks are distributed across different disks.

It is often convenient to express the complexity of external-memory algorithms using a number of frequently occurring primitive operations. The simplest operation is *scanning*, which means reading a stream of records stored consecutively on secondary memory. In this case, it is trivial to exploit disk- and block-parallelism. The number of I/Os is $\Theta(\frac{N}{DB}) = \Theta(\frac{n}{D})$. We abbreviate this quantity with *scan*(N). Algorithms for external *sorting* fall into two categories: those based on the *merging* and those based on the *distribution* paradigm. It has been shown that external sorting can be done with $\Theta(\frac{N}{DB} \log_{M/B} \frac{N}{B}) = \Theta(\frac{n}{D} \log_m n)$ I/Os. We abbreviate this quantity with *sort*(N).

3 External A*

In the following we study how to extend external exploration in A* search. The main advantage of A* with respect to other optimal exploration algorithms like breadth-first search or admissible depth-first search is that it traverses a smaller part of the search space to establish an optimal solution. In A*, the merit for state u is $f(u) = g(u)+h(u)$, with g being the cost of the path from the initial state to u and $h(u)$ being the estimate of the remaining cost from u to the goal. The new value $f(v)$ of a successor v of u is $f(v) = g(v) + h(v) = g(u) + w(u,v) + h(v) = f(u) + w(u,v) - h(u) + h(v)$. We first assume an undirected unweighted state space problem graph, and a *consistent* heuristic, where for all u and v we have, $w(u,v) \geq h(u) - h(v)$. These restrictions are often met in AI search practice. In this case we have $h(u) \leq h(v) + 1$ for every state u and every successor v of u. Since the problem graph is undirected this implies $|h(u) - h(v)| \leq 1$ so that $h(v) - h(u) \in \{-1, 0, 1\}$. This implies that the evaluation function f is monotonic non-decreasing. No successor will have a smaller f-value than the current one. Therefore, the A* algorithm, which traverses the state set in f-order, does not need to perform any *re-opening* strategy.

*External A** [5] maintains the search horizon on disk. The priority queue data structure is represented as a list of buckets ordered first by their $h + g$ value and then by the g value. In the course of the algorithm, each bucket *Open*(i,j) will contain all states u with path length $g(u) = i$ and heuristic estimate $h(u) = j$. As same states have same heuristic estimates it is easy to restrict duplicate detection to buckets of the same h-value. By an assumed undirected state space problem graph structure we can restrict aspirants for duplicate detection furthermore. If all duplicates of a state with g-value i are removed with respect to the levels i, $i-1$ and $i-2$, then there will remain no duplicate state for the entire search process. We consider each bucket for the *Open* list as a different file that has an individual internal buffer. A bucket is *active* if some of its states are currently expanded or generated. If a buffer becomes full then it is flushed to disk. Fig. 1 depicts the pseudo-code of the *External A** algorithm. The algorithm maintain the two values $g_{\min}$ and $f_{\min}$ to address the correct buckets. The buckets of $f_{\min}$ are traversed for increasing $g_{\min}$ unless the $g_{\min}$ exceeds $f_{\min}$. Due to the increase of the $g_{\min}$-value in the $f_{\min}$ bucket, an active bucket is closed when all its successors

```
**Procedure** *External A**
    *Open*$(0, h(\mathcal{I})) \leftarrow \{\mathcal{I}\}$
    $f_{\min} \leftarrow h(\mathcal{I})$
    **while** $(f_{\min} \neq \infty)$
        $g_{\min} \leftarrow \min\{i \mid Open(i,j) \neq \emptyset, i+j = f_{\min}\}$
        $h_{\max} \leftarrow f_{\min} - g_{\min}$
        **while** $(g_{\min} \leq f_{\min})$
            **if** $(h_{\max} = 0$ **and** $Open(g_{\min}, h_{\max})$ *contains terminal state* $u)$
                **return** *path*$(u)$
            $A(f_{\min}), A(f_{\min}+1), A(f_{\min}+2) \leftarrow succ(Open(g_{\min}, h_{\max}))$
            $A'(f_{\min}), A'(f_{\min}+1), A'(f_{\min}+2) \leftarrow$
                *remove duplicates from* $(A(f_{\min}), A(f_{\min}+1), A(f_{\min}+2))$
            $Open(g_{\min}+1, h_{\max}+1) \leftarrow A'(f_{\min}+2) \cup Open(g_{\min}+1, h_{\max}+1)$
            $Open(g_{\min}+1, h_{\max}) \leftarrow A'(f_{\min}+1) \cup Open(g_{\min}+1, h_{\max})$
            $Open(g_{\min}+1, h_{\max}-1) \leftarrow (A'(f_{\min}) \cup Open(g_{\min}+1, h_{\max}-1)) \setminus$
                                    $(Open(g_{\min}-1, h_{\max}-1) \cup Open(g_{\min}, h_{\max}-1))$
            $g_{\min} \leftarrow g_{\min} + 1$
        $f_{\min} \leftarrow \min\{i+j > f_{\min} \mid Open(i,j) \neq \emptyset\} \cup \{\infty\}$
```

Fig. 1. *External A** for consistent heuristics.

have been generated. Given $f_{\min}$ and $g_{\min}$ the corresponding h-value is determined by $h_{\max} = f_{\min} - g_{\min}$. According to their different h-values, successors are arranged into three different horizon lists $A(f_{\min})$, $A(f_{\min}+1)$, and $A(f_{\min}+2)$. Duplicate elimination is delayed.

Since *External A** simulates A* and changes only the order of elements to be expanded that have the same f-value, completeness and optimality are inherited from the properties of A*. The I/O complexity for External A* in an implicit unweighted and undirected graph with a consistent estimates is bounded by $O(sort(|E|) + scan(|V|))$, where $|V|$ and $|E|$ are the number of nodes and edges in the explored subgraph of the state space problem graph. If we additionally have $|E| = O(|V|)$, the complexity reduces to $O(sort(|V|))$ I/Os.

We establish the solution path by backward chaining from starting with the target state. There are two main options. Either we store predecessor information with each state on disk or, more elegantly, we for a state in depth g intersect the set of possible predecessors with the buckets of depth $g-1$. Any state that is in the intersection is reachable on an optimal solution path, so that we can recur. The time complexity is bounded by the scanning time of all buckets in consideration and surely in $O(scan(|V|))$. It has been shown [5] that the lower bound for the I/O complexity for delayed duplicate bucket elimination in an implicit unweighted and undirected graph A* search with consistent estimates is at least $\Omega(sort(|V|))$.

4 General Graphs

So far, we have looked at uniformly weighted graphs only. However, in practical model checking, the transition systems that are encountered are often directed. Also, validation of softwares and communication protocols often contains atomic regions. Atomic region corresponds to a block of statements that should be executed without the intervention of any other process. Atomic regions are represented in the general state graph as arcs with weights equal to the number of instructions in the atomic region. This motivates the generalization of graph search algorithms for non-uniformly weighted directed graphs. Later in this section, we discuss the effect of introducing heuristics in these algorithms.

We define, $w : E \rightarrow I\!R$ as the weight function for edges; the *weight* or *cost* of a path $p = (s = v_0, \ldots, v_k = v)$ can then be defined as $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. Path p is called *optimal path* if its weight is minimal among all paths between s and v; in this case, its cost is called the *shortest path distance* $\delta(s, v)$. The *optimal solution path cost* is abbreviated as $\delta(s, T) = \min\{t \in T \mid \delta(s, t)\}$, with T being the set of target states.

4.1 Directed Graphs

As seen above, undirected and unweighted graphs require to look at one previous and one successor layer only. For directed graphs, the efficiency of external algorithms is dependent on the duplicate elimination scope or *locality of the search*. The locality of a directed search graph is defined as $\max\{\delta(s, u) - \delta(s, v), 0\}$ for all nodes u, v, with v being a successor of u. In other words, locality determines the *thickness* of the layer of nodes needed to prevent duplicates in the search. It has been analyzed in the context of the *breadth-first heuristic search*.

Theorem 1. *[24] The number of previous layers of the graph that need to be retained to prevent duplicate search effort is equal to the locality of the state space graph.*

As a consequence, in undirected unweighted graphs, the locality is one and we need to store the immediate previous layer only, to check for duplicates.

Lemma 1. *For undirected, but weighted graphs the locality is smaller than or equal to the maximum edge weight $C = \max_{e \in E} w(e)$ in the state space graph.*

Proof. By the triangle inequality for shortest paths, we have $\delta(x, y) + \delta(y, z) \geq \delta(x, z)$ for all nodes x, y, z in the state space graph. For all $v \in \mathit{succ}(u)$ we have

$$\delta(s, u) - \delta(s, v) = \delta(u, s) - \delta(v, s) \leq \delta(u, v) \leq w(u, v).$$

Lemma 2. *Let D be the cost of the largest cycle in the state space graph. Then the locality of the state space graph is smaller than D.*

Proof. Let $\delta(v, u)$ be the smallest cost to get back from v to u in the global state space graph with $v \in \mathit{succ}(u)$. We have that $\delta(s, u) - \delta(s, v) \leq \delta(v, u)$ using the triangular property of shortest paths, so that $\max_{u, v \in \mathit{succ}(u)}\{\delta(s, u) - \delta(s, v)\} \leq \max_{u, v \in \mathit{succ}(u)} \delta(v, u) < D$.

In model checking, the global transition system is often composed of smaller components, called devices, processes or threads. For example, in the verification of software [1], the state space consists of the cross product of local state spaces, together with some additional information, e.g., on communication queues or global variables. As the product of the local state spaces is asynchronous, only one transition in one process is executed at a time.

Using asynchronism, we have that $\max_{u,v \in \textit{succ}(u)} \delta(v,u)$, in the global state space graph is bounded from below by $\max_{u',v' \in \textit{succ}(u')} \delta(v',u')$ in any local state space graph. Each cycle in the global state space is actually consists of local cycles in the local state spaces.

Lemma 3. *Let $p_1, \ldots, p_k$ be the processes in the system. If we denote D_i as the cost of the largest cycle in the graph representation of process p_i, $i \in \{1, \ldots, k\}$. Then we have that the locality is bounded by $D_1 + \ldots + D_k$.*

Proof. Let c be the largest cycle in the global state space graph with $w(c) = D$. As it decomposes into cycles $c_1 \ldots, c_k$ in the local state space graphs of the processes $p_1, \ldots, p_k$, we have that $D \leq w(c_1) + \ldots + w(c_k) \leq D_1 + \ldots + D_k$.

Even if the number of stored layers $b \geq 2$ is less than the locality of the graph, the number of times a node can be re-opened in breadth-first search is only linear in the depth of the search. This contrasts the exponential number of re-openings for linear-space depth-first search strategies.

Theorem 2. *[24] The worst-case number of times a node u can be re-opened is bounded by $\lfloor (\delta(s,T) - \delta(s,u))/b \rfloor$.*

If the locality is lesser than b, a breadth-first search algorithm does not need to consider re-openings of nodes at all.

4.2 Positive Weights

To compute the shortest path in weighted graphs, Dijkstra [4] proposed a greedy search strategy based on the *principle of optimality* $\delta(s,v) = \min_{v \in \textit{succ(u)}} \{\delta(s,u) + w(u,v)\}$. That is, the minimum distance from s to v is equal to the minimum of the sum of the distance from s to a predecessor u of v, plus the edge weight between u and v. This equation implies that any sub-path of an optimal path is itself optimal.

The exploration algorithm maintains an estimate of the minimum distance, more precisely, an upper bound $f(u)$ on $\delta(s,u)$ for each node u; initially set to ∞, $f(u)$ is successively decreased until it is equal to $\delta(u,v)$. From this point on, it remains constant throughout the rest of the algorithm. As the exploration of the problem graph is *implicit*, we additionally maintain a list *Closed* to store expanded nodes. The node relaxation procedure for a single-state algorithm, as opposed to the algorithms that work on sets of states, is shown in Fig. 2.

The correctness argument of the algorithm is based on the fact that for a node u with minimum f-value in *Open*, f is *correct*, i.e., $f(u) = \delta(s,u)$. Hence, when a node $t \in T$ is selected for removal from the $Open$, we have $f(t) = \delta(s,T)$. Moreover, if the weight function of a problem graph is strictly positive and if the weight of every infinite path is infinite, then Dijkstra's algorithm terminates with an optimal solution.

```
Procedure Relax
    if (Search(Open, $v$))
        if ($f(u) + w(u,v) < f(v)$)
            DecreaseKey(Open, $v$, $f(u) + w(u,v)$)
    else
        if not (Search(Closed, $v$))
            Insert(Open, $v$, $f(u) + w(u,v)$)
```

Fig. 2. Node relaxation in implicit Dijkstra's algorithm.

4.3 Re-Weighting

A heuristic h can be incorporated into Dijkstra's algorithm by a *re-weighting* transformation of the implicit search graph. The re-weighting function $\hat{w}(u,v)$ is defined as $\hat{w}(u,v) = w(u,v) - h(u) + h(v)$. If the heuristic is not consistent, re-weighting introduces negative weights into the problem graph. It is not difficult to obtain the following result.

Theorem 3. *Let $G = (V, E, w)$ be a weighted graph, and $h : V \to I\!R$ be a heuristic function. Let $\delta(s,t)$ be the length of the shortest path from s to t in the original graph and $\hat{\delta}(s,t)$ be the corresponding value in the re-weighted graph.*

1. *We have $w(p) = \delta(s,t)$, if and only if $\hat{w}(p) = \hat{\delta}(s,t)$, i.e., if p is the shortest path in the original graph, then p is also the shortest path in the re-weighted graph.*
2. *G has no negative weighted cycles with respect to w if and only if it has none with respect to $\hat{w}$.*

Proof. For proving the first assertion, let $p = (s = v_0, \ldots, v_k = t)$ be any path from the start node s to a goal node t. We have $\hat{w}(p) = \sum_{i=1}^{k} (w(v_{i-1}, v_i) - h(v_{i-1}) + h(v_i)) = w(p) - h(v_0)$. Assume that there is a path p' with $\hat{w}(p') < \hat{w}(p)$ and $w(p') \geq w(p)$. Then $w(p') - h(v_0) < w(p) - h(v_0)$; resulting in $w(p') < w(p)$, a contradiction. The other direction is dealt with analogously.

For the second assertion, let $c = (v_0, \ldots, v_l = v_0)$ be any cycle in G. Then we have $\hat{w}(c) = w(c) + h(v_l) - h(v_0) = w(c)$. □

The equation $h(u) \leq h(v) + w(u,v)$ is equivalent to $\hat{w}(u,v) = h(v) - h(u) + w(u,v) \geq 0$. Hence, a consistent heuristic yields a first A* variant of the algorithm of Dijkstra. It sets $f(s)$ to $h(s)$ for the initial node s and updates $f(v)$ with $f(u) + \hat{w}(u,v)$ instead of $f(u) + w(u,v)$ each time a node is selected. Since the shortest path p_t remains invariant through re-weighting, if $t \in T$ is selected from *Open*, we have

$$f(t) = \hat{\delta}(s,t) + h(s) = \hat{w}(p_t) + h(s) = w(p_t) = \delta(s,t).$$

4.4 Graphs with Edges of Negative Weight

Unfortunately, Dijkstra's algorithm fails on graphs with negative edge weights. As a simple example consider the graph consisting of three nodes with $w(s,u) = 4$,

```
**Procedure Relax**
    **if** (*Search*($Open, v$))
        **if** ($f(u) + w(u, v) < f(v)$)
            *DecreaseKey*($Open, v, f(u) + w(u, v)$)
    **else if** (*Search*($Closed, v$))
        **if** ($f(u) + w(u, v) \leq f(v)$)
            *Delete*($Closed, v$)
            *Insert*($Open, v, f(u) + w(u, v)$)
    **else**
        *Insert*($Open, v, f(u) + w(u, v)$)
```

Fig. 3. An node relaxation with re-opening that copes with negative edge weight.

$w(s, v) = 5$, and $w(v, u) = -2$, for which the algorithm of Dijkstra computes $\delta(s, u) = 4$ instead of the correct value $\delta(s, u) = 3$. The problem can be dealt with by *re-opening* already expanded node. The corresponding node relaxation procedure is shown in Fig. 3.

The following result was shown in the context of *route planning* [8], and is fundamental to prove the correctness of A* derivate.

Theorem 4. *[8] Let $G = (V, E, w)$ be a weighted graph and f be the cost of the shortest path so far for a particular node from the start node s in the modified algorithm of Dijkstra. At each selection of a node u from* Open, *we have the following invariant: Let $p = (s = v_0, \ldots, v_n = t)$ be a least-cost path from the start node s to a goal node $t \in T$. Application of* Relax *preserves the following invariant:*

(I) *Unless v_n is in* Closed *with $f(v_n) = \delta(s, v_n)$, there is a node v_i in* Open *such that $f(v_i) = \delta(s, v_i)$, and no $j > i$ exists such that v_j is in* Closed *with $f(v_j) = \delta(s, v_j)$.*

In short, at least we can be certain that there is one *good* node with perfect node evaluation in the *Open* list, that can be extended to an optimal solution path.

To optimize secondary storage accesses, expansions can be performed more efficiently if a particular order is selected. Invariant (I) is not dependent on the order that is present in the *Open* list.

In Fig. 4 we give a pseudo-code implementation for the node-ordering scheme. In contrast to Dijkstra's algorithm, reaching the first goal node will no longer guarantee optimality of the established solution path. Hence, the algorithm has to continue until the *Open* list runs empty. By storing and updating the current best solution path length as a global upper bound value α, it improves the solution quality over time.

Admissible heuristics are lower bounds, i.e. $h(u) \leq \delta(u, T)$ in the original graph. This corresponds to $0 \leq \hat{\delta}(u, T)$ in the re-weighted graph.

Theorem 5. *[8] If $\delta(u, T) \geq 0$, then the general node-ordering algorithm is optimal.*

```
**Procedure Node-Ordering**
    *Closed* $\leftarrow \{\}$; *Open* $\leftarrow \{s\}$;
    $\alpha \leftarrow \infty$; *best* $\leftarrow \emptyset$
    **while** (*Open* $\neq \emptyset$)
        $u \leftarrow$ *Select*(*Open*)
        *Open* $\leftarrow$ *Open* $\setminus \{u\}$; *Closed* $\leftarrow$ *Closed* $\cup \{u\}$
        **if** $(f(u) > \alpha)$ **continue**
        **if** (*terminal*$(u)$ **and** $f(u) < \alpha$)
            $\alpha \leftarrow f(u)$; *best* $\leftarrow$ *path*$(u)$
        **else**
            **for all** $v$ **in** *succ*$(u)$
                *Relax*$(u, v)$
    **return** *best*
```

Fig. 4. Relaxing the node expansion order.

But in model checking practice, we observe that non-admissible heuristics could appear. For example, the seemingly admissible heuristic *Active Processes* that for a given state identifies the number of active processes turned out to be non-admissible for some domains. Let's take the example of dining philosphers with deadlock detection. Assume that there are 2 philosophers A and B and both are thinking. This gives the number of active processes as 2. Now, A picks up her/his right fork. Since the left fork is still on the table, both A and B are still non-blocked. For the second move, let's assume that B picks up her/his right fork. This move blocks both A and B; resulting in the sudden decrease of number of active processes from 2 to 0. A heuristic is said to be admissible if it never overestimates the actual path cost. Here, with just one move we reached the deadlock as apposed to the heuristic estimate 2, implying that the heuristic was non-admissible.

As we saw that there are non-admissible heuristics, used in model checking, the re-weighted graph we will have $\delta(u, T) < 0$, implying that we cannot apply the above theorem. To further guarantee cost optimality of the solution, we have to extend the pruning criterion. It is not difficult to show that if we drop the criterion "*if* $(f(u) > \alpha)$ **continue**" then the algorithm is optimal for all re-weighted graph structures. We can prove slightly more.

Theorem 6. *If we set $f(u)+\delta(u, T) > \alpha$ as the pruning condition in the node ordering algorithm, then the algorithm is optimal.*

Proof. Upon termination, each node inserted into *Open* must have been selected at least once. Suppose that invariant (I) is preserved in each loop, i.e., there is always a node v in the *Open* list on an optimal path with $f(v) = \delta(s, v)$. Thus the algorithm cannot terminate without eventually selecting the goal node on this path, and since by definition, it is not more expensive than any found solution path and *best* maintains the currently shortest path, an optimal solution will be returned. It remains to show that the invariant (I) holds in each iteration. If the extracted node u is not equal to v there is nothing to show. Otherwise $f(u) = \delta(s, u)$. The bound α denotes the currently best solution length. If $f(u) + \delta(u,T) \leq \alpha$ no pruning takes place. On the other hand $f(u) + \delta(u,T) > \alpha$ leads to a contradiction since $\delta(s,T) = \delta(s,u) + \delta(u,T) = f(u) + \delta(u,T) > \alpha \geq \delta(s,T)$. □

Unfortunately, we do not know the value of $\delta(s,T)$, so the only thing that we can do is to take a lower bound to it. Since h that has been used earlier on is not admissible, we need a different bound or condition. For the original graph, it is easy to see that all nodes that have a larger path cost value than the obtained solution path cannot lead to a better solution, since the weights in the original graph are non-negative. Consequently, if $g(u)$ denotes the path length from s to u, $g(u) > \alpha$ is one pruning condition that we can apply in the original graph.

5 Explicit-State Model Checking in SPIN and HSF-SPIN

SPIN [10] is probably the most prominent explicit state model checking tool. Models are specified in its input language Promela. The language is well-suited to specify communication protocols, but has also been used for a wide range of other verification tasks. The model checker transforms the input into an internal automata representation, which, in turn, is enumerated by its exploration engine. Several efficiency aspects ranging from partial-order reduction to bit-state hashing enhance the exploration process. The parser produces sources that encode states and state transitions in native C code. These are linked together with the validation module to allow exploration of the model. The graphical user interface XSPIN allows to code the model, run the validator, show the internal automata representation, and simulate traces with message sequence charts.

Our own experimental model checker HSF-SPIN [6] is a compatible extension to SPIN. Additionally it incorporates directed search in explicit state model checking. The tool has been designed to allow different search algorithms by providing a general state expanding subroutine. In its current implementation it provides depth-first and breadth-first search as well as heuristic search algorithms like best-first search, A* and IDA*, and local search algorithms like hill-climbing and genetic algorithms. Partial order and symmetry reduction have been successfully combined with this portfolio [16, 15]. HSF-SPIN can handle a significant fraction of Promela and deals with the same input and output formats as SPIN. Heuristic search in HSF-SPIN combines positively with automated abstractions in form of abstraction databases.

The internal representation of a state consists of two parts. The first part contains information necessary for the search algorithms. This includes the estimated value for the state to the goal, the cost of the current optimal path to the state, a link to the predecessor state and information about the transition that lead to the state. The second part contains the representation of the state of the system and is usually called state vector. This part is represented similarly as in SPIN. Few modifications were, however, necessary due to technical details. Basically, the state vector contains the value of the variables and the local state of each process.

The expansion function is a fundamental component of the verifier. Actually, it was the component of HSF-SPIN that required most of the implementation efforts. It takes the representation of a state as input and returns a list containing each successor state.

The use of this function in each search algorithm implies that the implementation of the depth-first search is not the most efficient.

All heuristic functions return a positive integer value for a given state. Some of them profit from information gathered before the verification starts. For example, the FSM distance estimate requires to run the all-pairs shortest path algorithm on the state transition graph of each process type. On the other hand, the deadlock inferring approach allows the user to determine explicitly which states have to be considered as potentially blocking by labeling statements in the model.

6 From HSF-Spin to IO-HSF-SPIN

Although theoretically simple, the practical extension of an existing explicit model checker like SPIN to external search is not trivial, and poses a lot of subtle implementation problems. In an external model it is required that the algorithm should be capable of writing any intermediate result to the disk, reading it again at any point of time in the future, and reusing it like it remained in the main memory. This requirement turned out to be a non-trivial one in order to adapt SPIN for external model checking. As described above, SPIN's state consists of two parts: state's information and the state vector. The state vector can be viewed as a sequence of active processes and message queues, describing the actual state of the system being checked.

SPIN is highly optimized for efficiency and hence uses a lot of global variables. These global variables are used to store the meta information about the state vector. This meta information consists of the address information of processes and queues in the state vector. Since the information about actual addresses would be void once a state has been flushed to the disk and retrieved back to a new memory address, we suggested to save the information that can *reset* the global variables to work on the new location of the state vector. We identified that with the order and type of each element in the state vector in hand, we can reset all the global variables. This motivates us to extend the state's description.

The new state's description $\mathcal{S}$ can be viewed as a 4-tuple, $(M, \sigma, \kappa, \tau)$, where M is the information about the state, e.g., its g and h values, size of the state vector, etc., σ is the state vector, and κ can be defined as $\kappa : \sigma \rightarrow \{Process,\ Queue\}$, i.e., given an element $\sigma_i \in \sigma$, κ identifies whether σ_i is a *Process* or a *Queue*. SPIN differentiates between different types of processes (resp. queues) by assigning an ID to each of them. If $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ is the set of all processes and $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_m\}$ is the set of all queues, $\tau : \sigma_i \in \sigma \rightarrow \mathcal{P}$, if $\kappa(\sigma_i) = Process$ or $\tau : \sigma_i \in \sigma \rightarrow \mathcal{Q}$, otherwise.

We employed a two level memory architecture for storing the states. Initially all states are kept in the internal memory. An upper limit is defined on the maximum consumption of the internal memory available. If the total memory occupied by the stored states exceeds the maximum limit, the external memory management routines are invoked that flushes the excess states to the disk. The advantage of having a two-level memory management routine is to avoid the I/Os when the internal memory is sufficiently large for a particular problem.

A $bucket(i, j)$ is represented internally by a fixed size buffer. When the buffer gets full, it is sorted and flushed to the disk by appending it at the end of the corresponding file. Duplicates removal is then done in two stages. First, an external merge and compaction of the sorted flushed buffers that removes all the duplicates in the file is performed. Second, the files of the top layers with the same h values but smaller g values are subtracted from the resulting file. The number of top layers that are checked depends on the locality of the graph, as explained earlier.

Consequently, the reading of states for expansion is done by copying a part of the external bucket file to the corresponding internal buffer. Once the whole buffer is scanned and the successors are generated for all the states within the buffer, the next part of the file is read in the same buffer. The process continues until the file is completely read.

7 Experiments

N	d	s	e	t	Space (in gigabytes)
100	402	980,003	19,503	999,504	2.29
150	603	3,330,003	44,253	3,374,254	10.4

Table 1. Deadlock Detection in Dining Philosophers

N	d	s	e	t	Space (in gigabytes)
5	33	10,874	4,945	24,583	0.0038
7	45	333,848	115,631	820,319	0.133
8	50	420,498	103,667	917,011	0.182
9	57	9,293,203	2,534,517	23,499,519	4.29

Table 2. Deadlock Detection in Optical Telegraph

We choose three classical and challenging protocol models namely, dining philosophers, optical telegraph and CORBA-GIOP for our experiments. The property to search for is the *deadlock* property. We employed the number of active processes as the heuristics to guide the exploration. For each experiment we report, the solution depth d, number of stored nodes s, number of expansions e, number of transitions t, and the space requirement of the stored nodes $Space$. The experiments are performed on a 4 processors Sun Ultra Sparc running Solaris operating system and using GCC 2.95 as the compiler. Additionally, symmetry reduction is employed in all experiments.

Table 1 presents the results for the deadlock detection for different instances of dining philosophers problem. The bottleneck in the dining philosopher's problem is not only the combinatorial explosion in the number of states but also the size of the states . As can be observed in the last column depicting the space requirement, the problem instance with 150 philosophers requires a storage space of 10.2 gigabytes, which is much higher than even the address limits of present micro computers.

The second domain is the optical telegraph model. We conducted experiments with different number of stations. The results are presented in Table 2.

N	M	d	s	e	t	Space (in gigabytes)
2	1	58	48,009	39,260	126,478	0.033
3	1	70	825,789	670,679	2,416,823	0.572
4	1	75	7,343,358	5,727,909	22,809,278	5.17
2	2	64	158,561	125,514	466,339	0.121
3	2	76	2,705,766	2,134,724	8,705,588	2.1
4	2	81	26,340,417	20,861,609	88,030,774	20.7

Table 3. Deadlock Detection in CORBA - GIOP

CORBA - GIOP [12] turned out to be one of the hardest models to verify because of its enormous state space. The reason is the high branching factor in the state space graph that results in the generation of a large number of states. The model takes two main parameters namely: the number of users N and the number of servers M with a range restriction of 1 to 4 on N and 1 to 2 on M. We have been able to solve all the instances of the GIOP model especially the configuration with 4 users and 2 servers which requires a storage space of 20.7 gigabytes.

One of the main hurdles while running the experiments was the system limit on the number of file pointers that can be opened at a particular time. A large number of file pointers are the requirements while merging the sorted flushed buffers. For the files that needed file pointers more than the system limit, the experiments are re-run with larger internal buffer size that results in smaller number of large sorted buffers.

Summing up, the largest problem size reported to be solved by the first external model checker Murϕ [23] consisted of 1,021,464 states of 136 bytes each. This gives the overall space requirement of 0.129 gigabytes. With the presented approach, the largest problem size that we have been able to solve requires 20.7 gigabytes.

8 Conclusions

With this article we contribute the first theoretical and analytical study of I/O efficient directed model checking. In the theoretical part of the paper, we extended External A* to directed and weighted graphs. Through the process of re-weighting, we refer to some general results of node-ordering in the exploration of graphs with negative weights. We give different pruning conditions for node ordering algorithms for admissible and non-admissible heuristics. Moreover, we showed some necessary conditions on the locality of the graph to ease duplicate detection during the exploration. The concepts are then extended to the situation in model checking, where the global state space is composed of local transition graphs.

In the practical part of the paper, we have seen a non-trivial implementation to extend a state-of-the-art model checker SPIN to allow directed and external search. The first results on challenging benchmark protocols show that the external algorithms reduce the memory consumption, are sufficiently fast in practice, and have some further advantages by using a different node expansion order. It should be noted here that some of the hardest problems like GIOP and dining philosophers with scaled parameters that were not tractable earlier due to the internal memory requirements, have been solved for the first time. The present implementation is capable of coping with negative edges in the presence of inconsistent heuristic.

9 Related and Future Work

We are, however, not the first ones that look at the performance of model checkers on external devices. One of the first approaches toward this direction was proposed [23] in the context of the Murϕ validation tool. With a special algorithm disk instead of main memory is used for storing almost all of the state table at the cost of a small runtime penalty, which is typically around 15% when the memory savings factor is between one and two orders of magnitude. The algorithm linearizes the accesses to the state table and amortizes the cost of accessing the whole table over all the states in a breadth-first search level.

External breadth-first search for explicit graph structures that reside on disk has been introduced by [19]. It was improved to a sub-linear number of I/Os in [17]. Single-source shortest-pair algorithms that deal with explicit graphs stored on disk are surveyed in [18]. Delayed duplicate detection [14] adapts external BFS to implicit graphs. This extension to the External A* algorithm as proposed in [5] exploits the work of [24]. Zhou and Hansen [25] also worked on a different solution to apply external search for AI domains. They term their approach as *structured duplicate detection*, in which they use state space abstraction to establish which part of the search space can be externalized, and which part cannot.

Korf [13] also successfully extended delayed duplicate detection to best-first search and considered omission of the visited list as proposed in *frontier search*. In his proposal, it turned out that any 2 of the 3 options are compatible yielding the following set of algorithms: *breadth-first frontier search with delayed duplicate detection*, *best-first frontier search*, and *best-first search with external non-reduced closed list*. In the last case, the algorithm simulate a buffered traversal in a bucket-based priority queue. With External A* it turns out that one can combine all three approaches. In Korf's work, external sorting based on hash function partition is proposed. In summary, all external AI search approaches have independent contributions and can cooperate.

There is a tight connection between external and symbolic exploration algorithms. Both approaches consider sets of states instead of individual ones. It is apparent that the presented approach for an explicit state model checking will transfer to symbolic model checking. For example, in explicit graph theory, the external computation for the all-pair shortest path problem is studied in [22], while symbolic single-source and all-pair shortest path algorithms are considered in [20, 21]. As an interesting side effect, the symbolic algorithm of Bellman-Ford often outperformed Dijkstra's algorithm on symbolically represented graphs. For heuristic search, the splitting of the *Open*-list into buckets as seen in this text, corresponds to the fg-search method in the SetA* [11] version of the BDDA* algorithm [7]. However, refined considerations on the duplicate scope as presented here have not been studied. The implementation of an external directed search symbolic model checker is one of our future research goals.

Acknowledgments The authors are supported by DFG in the projects ED 74/2 and ED 74/3. Thousands thanks to Alberto Lluch-Lafuente at University of Pisa for many detailed discussions on his implementation of HSF-SPIN and its possible extension. We are also grateful to Stefan Schrödl, with whom we developed the *External A** algorithm, and who is jointly responsible for some of the basic observations for the development of the node-ordering algorithm.

References

1. B. Bérard, A. F. M. Bidoit, F. Laroussine, A. Petit, L. Petrucci, P. Schoenebelen, and P. McKenzie. *Systems and Software Verification.* Springer, 2001.
2. J. R. Burch, E. M.Clarke, K. L. McMillian, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.
4. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
5. S. Edelkamp, S. Jabbar, and S. Schroedl. External A*. In *German Conference on Artificial Intelligence (KI)*, pages 226–240, 2004.
6. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*, 2004.
7. S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, pages 81–92, 1998.
8. S. Edelkamp and S. Schrödl. Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.
9. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics*, 4:100–107, 1968.
10. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1990.
11. R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *National Conference on Artificial Intelligence (AAAI)*, 2002.
12. M. Kamel and S. Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000.
13. R. Korf. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, pages 650–657, 2004.
14. R. E. Korf. Breadth-first frontier search with delayed duplicate detection. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, pages 87–92, 2003.
15. A. Lluch-Lafuente. Symmetry reduction and heuristic search for error detection in model checking. In *Model Checking and Artificial Intelligence (MoChArt-03)*, 2003.
16. A. Lluch-Lafuente, S. Leue, and S. Edelkamp. Partial order reduction in directed model checking. In *Workshop on Model Checking Software (SPIN)*, pages 112–127, 2002.

17. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, pages 723–735, 2002.
18. U. Meyer, P. Sanders, and J. Sibeyn. *Memory Hierarchies*. Springer, 2003.
19. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 87–88, 2001.
20. D. Sawatzki. Experimental studies of symbolic shortest-path algorithms. In *30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, 2004. to appear.
21. D. Sawatzki. A symbolic approach to the all-pairs shortest-paths problem. In *Workshop on Experimental and Efficient Algorithms (WEA)*, pages 492–497, 2004.
22. J. F. Sibeyn. External matrix multiplication and all-pairs shortest path. *Information Processing Letters*, 91(2):99–106, 2004.
23. U. Stern and D. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *International Conference on Computer Aided Verification (CAV)*, pages 172–183, 1998.
24. R. Zhou and E. Hansen. Breadth-first heuristic search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 92–100, 2004.
25. R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI)*, pages 683–689, 2004.

Appendix: Proof of the Theorems

Theorem 1 The number of previous layers of the graph that need to be retained to prevent duplicate search effort is equal to the locality of the search graph.

Proof. To prove equality, we have to show that if the number of stored layers of a breadth-first search graph is smaller than the locality l, this can lead to re-openings, and if the number is greater than or equal to the locality, there are no re-openings.

For the first case, consider u and v with $\delta(s,u) - \delta(s,v) > l$. We will show that there is a duplicate node. When u is expanded, its successor v is either in the boundary of the previous k layers, in which case no re-openings occurs, or it is not, in which case, we have a re-opening of v. However, in the first case v has a sub-optimal depth and has been previously re-opened.

If the number of stored layers of a breadth-first search graph is greater than or equal to the locality of a graph this prevents re-openings as follows. Certainly, there is no re-opening in the first l layers. By induction, when a new layer is generated, no previously deleted node can be re-generated.

Theorem 2 The worst-case number of times a node u can be re-opened is bounded by $\lfloor(\delta(s,T) - \delta(s,u))/b\rfloor$.

Proof. We have no duplicate in every b layers. Therefore, the earliest level for a node u to be re-opened is $\delta(s,u)+b$ and the earliest next level it will be re-opened is $\delta(s,u)+2b$ and so on. Since the total number of layers is bounded by $\delta(s,T)$ the number of re-openings for a node cannot exceed $\lfloor(\delta(s,T) - \delta(s,u))/b\rfloor$.

Theorem 4 Let $G = (V,E,w)$ be a weighted graph and f be the cost of the shortest path so far for a particular node from the start node s in the modified algorithm of Dijkstra. At each selection of a node u from *Open*, we have the following invariant: Let $p = (s = v_0, \ldots, v_n = t)$ be a least-cost path from the start node s to a goal node $t \in T$. Application of *Relax* preserves the following invariant:

(I) Unless v_n is in *Closed* with $f(v_n) = \delta(s, v_n)$, there is a node v_i in *Open* such that $f(v_i) = \delta(s, v_i)$, and no $j > i$ exists such that v_j is in *Closed* with $f(v_j) = \delta(s, v_j)$.

Proof. Without loss of generality, let i be maximal among the nodes satisfying (I). We distinguish the following cases:

1. Node u is not on p or $f(u) > \delta(s, u)$. Then node $v_i \neq u$ remains in *Open*. Since no v in *Open* $\cap$ p $\cap\, \Gamma(u)$ with $f(v) = \delta(s, v) \leq f(u) + w(u, v)$ is changed and no other node is added to *Closed*, (I) is preserved.
2. Node u is on p and $f(u) = \delta(s, u)$. If $u = v_n$, there is nothing to show.
   First assume $u = v_i$. Then *Relax* will be called for $v = v_{i+1} \in \Gamma(u)$; for all other nodes in $\Gamma(u) \setminus \{v_{i+1}\}$, the argument of case 1 holds. According to (I), if v is in *Closed*, then $f(v) > \delta(s, v)$, and it will be reinserted into *Open* with $f(v) = \delta(s, u) + w(u, v) = \delta(s, v)$. If v is neither in *Open* or *Closed*, it is inserted into *Open* with this merit. Otherwise, the *DecreaseKey* operation will set it to $\delta(s, v)$. In either case, v guarantees the invariant (I).
   Now suppose $u \neq v_i$. By the maximality assumption of i we have $u = v_k$ with $k < i$. If $v = v_i$, no *DecreaseKey* operation can change it because v_i already has optimal merit $f(v) = \delta(s, u) + w(u, v) = \delta(s, v)$. Otherwise, v_i remains in *Open* with unchanged f-value and no other node besides u is inserted into *Closed*; thus, v_i still preserves (I). $\square$

Theorem 5 If $\delta(u, T) \geq 0$, then the node-ordering algorithm is optimal.

Proof. Upon termination, each node inserted into *Open* must have been selected at least once. Suppose that invariant (I) is preserved in each loop, i.e., that there is always a node v in the *Open* list on an optimal path with $f(v) = \delta(s, v)$. Thus the algorithm cannot terminate without eventually selecting the goal node on this path, and since by definition it is not more expensive than any found solution path and *best* maintains the currently shortest path, an optimal solution will be returned. It remains to show that the invariant (I) holds in each iteration. If the extracted node u is not equal to v there is nothing to show. Otherwise $f(u) = \delta(s, u)$. The bound α denotes the currently best solution length. If $f(u) \leq \alpha$ no pruning takes place. On the other hand $f(u) > \alpha$ leads to a contradiction since $\alpha \geq \delta(s, u) + \delta(u, T) \geq \delta(s, u) = f(u)$ (the latter inequality is justified by $\delta(u, T) \geq 0$). $\square$