Accelerating External Search with Bitstate Hashing

Stefan Edelkamp and Shahid Jabbar

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

Abstract. In this paper we refine external exploration for explicit state model checking by a fusion of External A* with internal bit state hashing. External A* provides a method to cope up with large state spaces by efficiently utilizing secondary storage devices like harddisk to maintain the open and closed lists. Duplicates are removed by a two-level refinement scheme that involves sorting a subset of the open list externally and subtracting a small subset of closed list from the open list.
The bottleneck in External A* [7] is the duplicates removal phase that dominates the I/O complexity of External A*. Bitstate hashing provides a solution to faster duplicates removal by utilizing only few bits for each state. But bitstate hashing is faced with the problem of having no support for large *open* list and for solution reconstruction.
We present a strategy to accelerate external search by using bitstate hashing for duplicates removal. Case studies with our experimental external explicit state model checker IO-HSF-SPIN illustrate the effectiveness of the proposed algorithms.
Keywords: External Model Checking, External Search, Bitstate hashing.

1 Introduction

In explicit-state model checking software [1], state descriptors are often so large, so that main memory is often not sufficient for a lossless storage of the set of reachable states during the exploration even if all available reduction techniques, like symmetry [9] or partial-order reduction [12] have been applied.

Besides advanced implicit storage structures for the set of states [8] three different options have been proposed to overcome the internal space limitations for this so-called *state explosion* problem, i.e. directed, partial and secondary search.

Directed or heuristic search [11] guides the search process into the direction of the goal states, which in model checking safety properties is the set of software errors. The main observation is that using this guidance, the number of explored states needed to establish an error is smaller than with blind search. Moreover, directed model checking often reduce the length of the counter-example, which in turn eases the interpretation of the bug.

*External A** corresponds to a variant of A* algorithm that utilizes hard-disk to store the *closed* and *open* lists during exploration. The I/O complexity of External A* algorithm for undirected and unweighted graphs, and employing consistent heuristic estimates accumulates to $O(scan(|V|) + sort(|E|))$ I/Os, where V and E are the set of nodes and edges in the graph respectively. Recently [7] extended the External A* algorithm for weighted and directed graphs as they appear in model checking.

Bitstate hashing [5] refers to storage of huge state sets in hash tables with only one or a few bits reserved for each state. Invoking such partial search option implies that a retrieved node might be an unexpected synonym, since there is no way to distinguish a real duplicate from an erroneous one. Reconsidering stored states, however, is not possible, as the information of generating path length and predecessor path length might be false. The lack of state space coverage in partial search is compensated by repeating the search with restarts on different hash functions.

Motivation: We address three problems in this paper.

1. The bottleneck in external directed model checking is the delayed duplicate detection phase that dominates the $O(scan(|V|) + sort(|E|))$ I/O complexity of the External A* algorithm.
2. Even though bitstate hashing provides a very space efficient duplicate detection scheme, searching with bitstate hashing is still restricted by the internal memory available needed to maintain the *open* list. As is generally the case that size of the *open* list can easily go beyond the main memory limit especially in heuristic guided explorations.
3. Since bitstate hashing uses only a few bits to represent a state, solution reconstruction is clearly out of question.

Solution: We propose the idea of extending the methodology presented in [7] for I/O efficient model checking, to utilize bitstate hashing for duplicates removal. We suggest to use an internal bitstate hash table to detect duplicates during the external search. Surprisingly, this fusion of ideas helps us to tackle all three problems. We see faster duplicate detection because of bitstate hashing, plenty of secondary storage to maintain larger *open* list, and undeleted *closed* list residing on the disk for solution reconstruction.

Structure: First, we briefly recall directed, partial search. Next we study External A* and its extension with internal bitstate hashing. We then present initial experimental results obtained with over external model checker IO-HSF-SPIN. Last but not least, we draw conclusions and discuss future work.

2 Directed Model Checking

Directed model checking [3] incorporates heuristic search algorithms like A* [11] to enhance the bug-finding capability of model checkers, by accelerating the search for errors and finding (near to) minimal counterexamples. In that manner we can mitigate the state explosion problem and the long counterexamples provided by some algorithms like DFS, which is often applied in explicit model checking.

One can distinguish different classes of evaluation functions based on the information they try to exploit. *Property specific* heuristics [3] analyze the error description as the negation of the correctness specification. In some cases the underlying methods are only applicable to special kinds of errors. A heuristic that prioritizes transitions that block a higher number of processes focuses on deadlock detection. In other cases the approaches are applicable to a wider range of errors. For instance, there are heuristics for invariant checking that extract information from the invariant specification and heuristics that base on already given errors states. The second class has been denoted as being *structural* [4], in the sense that source code metrics govern the search. This class includes coverage metrics (such as *branch count*) as well as concurrency measures (such as *thread preference* and *thread interleaving*). Next there is the class of *user heuristics* that inherit guidance from the system designer in form of source annotations, yielding preference and pruning rules for the model checker.

3 Partial Search

For large problem spaces, it is very space efficient to apply a depth-first search strategy in combination with duplicate detection via a membership data structure based on bit-vector. This is also the approach of the software model SPIN [6]. Let n be the number of reachable states and m be the maximal number of bits available. As a coarse approximation for *single bit-state* hashing with $n < m$, the average probability P_1 of a false-positive error during the course of the search is bounded by $P_1 \leq \frac{1}{n}\sum_{i=0}^{n-1} \frac{i}{m} \leq n/2m$, since the i-th element collides with one of the $i-1$ already inserted elements with a probability of at most $(i-1)/m$, $1 \leq i \leq n$. For multi-bit hashing using h (independent) hash-functions with the assumption $hn < m$, the average probability of collision P_h is reduced to $P_h \leq \frac{1}{n}\sum_{i=0}^{n-1}(h \cdot \frac{i}{m})^h$, since i elements occupy at most hi/m addresses, $0 \leq i \leq n-1$.

An attempt to remedy the incompleteness of partial search is to re-invoke the algorithm several times with different hash functions to improve the coverage of the search tree. This technique, called *sequential hashing*, successively examines various beams in the search tree (up to a certain threshold depth).

4 External Partial Directed Search

*External A** [2] maintains the search horizon on disk. The priority queue data structure is represented as a list of buckets. In the course of the algorithm, each bucket *Open*(i, j) will contain all states u with path length $g(u) = i$ and heurstic 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

```
Procedure External A*
    $Open(0, h(\mathcal{I})) \leftarrow \{\mathcal{I}\}$
    $\mathcal{H}[0 \ldots m-1]$: Bitstate Hashtable
    for $i = 0$ to $m - 1$
        $\mathcal{H}[i] \leftarrow$ false
    $f_{\min} \leftarrow h(\mathcal{I})$
    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 $path(u)$
            forall $v \in succ(Open(g_{\min}, h_{\max}))$
                if(not $\mathcal{H}[v]$)
                    $\mathcal{H}[v] \leftarrow$ true
                    $A'(f_{\min}), A'(f_{\min}+1), A'(f_{\min}+2) \leftarrow v$
            $Open(g_{\min}+1, h_{\max}+1) \leftarrow A'(f_{\min}+2) \cup Open(g_{\min}+1, h_{\max}+1)$
            $Open(g_{\min}+1, h_{\max}) \leftarrow A'(f_{\min}+1) \cup Open(g_{\min}+1, h_{\max})$
            $Open(g_{\min}+1, h_{\max}-1) \leftarrow (A'(f_{\min}) \cup Open(g_{\min}+1, h_{\max}-1))$
            $g_{\min} \leftarrow g_{\min} + 1$
        $f_{\min} \leftarrow \min\{i + j > f_{\min} \mid Open(i,j) \neq \emptyset\} \cup \{\infty\}$
```

Fig. 1. *External A** with internal bitstate hashing.

removed with respect to the levels i, $i-1$ and $i-2$, then there no duplicate state will remain for the entire search process. For breadth-first-search in explicit graphs, this is in fact the algorithm of [10]. 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 being expanded or generated. If a buffer becomes full, then it is flushed to the corresponding bucket file on the disk. The algorithm maintains two values $g_{\min}$ and $f_{\min}$ to address the correct buckets. The buckets of $f_{\min}$ are traversed for increasing $g_{\min}$-value 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 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 different horizon lists. 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, and $scan(n)$ $(sort(n))$ are the number of I/O needed to scan (sort) n elements.

It has been shown [2] 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 $\Omega(sort(|V|))$.

In model checking we deal with directed and weighted graphs. External A* has been extended in [7] to deal with such graphs. The introduction of directed edges effects the *locality of the search*, which dictates the number of previous layers that have to be subtracted for duplicate detection. In directed and weighted graphs, the locality is bounded by the weight of the largest cycle in the graph.

In Figure 1, we see the proposed algorithm in pseudo-code form. The major change from the original algorithm [2] is the replacement of delayed duplicate detection by bitstate hashing. A hashtable $\mathcal{H}$ is used that checks for the existence of duplicates for each successor obtained by the expansion of the bucket $Open(g_{\min}, h_{\max})$. With the variable $\mathcal{I}$ we represent the initial state. The open list for a particular diagonal f is represented with $A(f)$.

5 Experiments

For experimental validation of the proposed idea, we have extended our experimental external directed model checker IO-HSF-SPIN to utilize internal bitstate hashing. All experiments are performed on a Linux Pentium IV machine with 2 GB of internal memory and 2 harddisks of 120 Gigabytes each, combined with disk-stripping technique (RAID level 0).

The experiments are performed on three protocols namely optical telegraph, CORBA-GIOP protocol, and dinning philosophers. The existence of a deadlock is searched in all the instances, using the number of active processes in each state as the heuristic estimate. Moreover, partial order reduction was not employed.

N	d	s	s_b	t (mm:ss)	t_b (mm:ss)
7	45	333,877	243,318	00:14.47	00:07.39
8	50	420,531	332,963	00:22.25	00:11.20
9	57	9,186,611	6,350,087	17:26.88	04:10.33

Table 1. Deadlock Detection in Optical Telegraph

We compare the runnning times of both approaches along with the number of stored states. In the following tables, d denotes the depth of the error state, s and s_b denote the number of stored states by the delayed duplication detection and by internal bitstate hashing, respectively. Running times are reported in the columns t and t_b, respectively corresponding to the run-time with delayed duplicate detection and with internal bitstate hashing. Furthermore, in Table 1 the parameter N denotes the number of stations. Similarly, in Table 2, N corresponds to the number of clients and M to the number of servers. Column N in Table 3 refers to the number of philosophers.

A gain in the running time is easily observable. For some of the instances, we notice that the number of stored states by both the approaches do not differ much, which points to the small approximation factor due to bitstate hashing.

N	M	d	s	s_b	t (mm:ss)	t_b (mm:ss)
2	1	58	52,410	48,000	00:11.36	00:03.49
3	1	70	893,392	809,098	03:03.79	00:43.26
4	1	75	7,929,710	6,998,693	22:49.23	08:43.01
3	2	76	3,431,619	3,127,718	05:43.16	02:42.54
4	2	81	30,504,630	27,133,946	293:00.00	33:06.28

Table 2. Deadlock Detection in CORBA - GIOP

The hardest instance of CORBA-GIOP (cf. Table 2) is the one with 4 clients and 2 servers. An earlier solution reported in [7] took about 20 gigabytes of harddisk storage to solve this problem.

N	d	s	s_b	t (mm:ss)	t_b (mm:ss)
100	402	999,810	975,459	01:05	01:39
150	603	3,330,456	3,319,312	05:44	06:18

Table 3. Deadlock Detection in Dining Philosophers

For the sake of completeness in our exposition, in Table 3 we present a special case. Here we observe an increase in time with bitstate hashing. A profiling of the execution revealed that due to large state sizes (ca. 2000 bytes), computation of hash value became the dominating factor of the running time. Note that the difference between the timings is lesser for the 150 philosophers case as with the 100 philosophers case. Ideally, this difference should go down towards negative with increase in disk activity.

6 Conclusion

The paper contributes the first study of combining external and partial search. As bit-state hash tables are state-of-the-art in saving memory with internal model checking, we studied its extension to algorithms involving secondary memory. The gains are three folds, namely, faster duplicate detection due to internal bitstate hashing, support for very large *open* lists due to external storage, and possibility of solution reconstruction due to persistent *closed* lists.

The presented approach has been implemented on top of our experimental model checker IO-HSF-SPIN. We observed a gain in the run-time of External A* algorithm, going up to 4 times for some of the cases. For smaller problems the gain is not very substantial. The reason is the extra overhead of hash computation that dominates the running time and sometimes surpasses the time needed to do the delayed duplicate detection.

We hope that the applicability of the current approach will help to push the limits of practical model checking. In future we plan to extend our methodology to large bitstate hash tables that cannot fit into the main memory.

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. S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In S. Biundo, T. Frühwirth, and G. Palm, editors, *KI 2004: Advances in Artificial Intelligence: 27th Annual German Conference on AI*, volume 3238 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 226–240. Springer-Verlag, 2004.
3. 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*, 5(2–3):247–267, 2004.
4. A. Groce and W. Visser. Heuristic model checking for java programs. *International Journal on Software Tools for Technology Transfer*, 6(4), 2004.
5. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.
6. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
7. S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science (LNCS)*, pages 313–329. Springer-Verlag, 2005.
8. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.
9. K. L. McMillan. Symmetry and model checking. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 117–137. Springer-Verlag, 1998.
10. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 87–88, 2001.
11. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
12. D. A. Peled. Ten years of partial order reduction. In *Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science (LNCS)*, pages 17–28. Springer-Verlag, 1998.