Alcom-FT Technical Report Series
ALCOMFT-TR-02-51

Theory and Practice of Time-Space Trade-Offs in Memory Limited Search

Stefan Edelkamp[1] and Ulrich Meyer[2]

[1] Institut für Informatik
Georges-Köhler-Allee, Geb. 51
79110 Freiburg, Germany
edelkamp@informatik.uni-freiburg.de
[2] Max-Plank-Institut für Informatik
Stuhlsatzenhausenweg 85
66123 Saarbrücken, Germany
umeyer@mpi-sb.mpg.de

Abstract. Having to cope with memory limitations is an ubiquitous issue in heuristic search. We present theoretical and practical results on new variants for exploring state-space with respect to memory limitations.
We establish $O(\log n)$ minimum-space algorithms that omit both the open and the closed list to determine the shortest path between every two nodes and study the gap in between full memorization in a hash table and the information-theoretic lower bound. The proposed structure of suffix-lists elaborates on a concise binary representation of states by applying bit-state hashing techniques. Significantly more states can be stored while searching and inserting n items into suffix lists is still available in $O(n \log n)$ time. Bit-state hashing leads to the new paradigm of partial iterative-deepening heuristic search, in which full exploration is sacrificed for a better detection of duplicates in large search depth. We give first promising results in the application area of communication protocols.

1 Introduction

Heuristic search in large problem spaces inherently calls for algorithms capable of running under restricted memory. We present new data structures and algorithms that face the memory vs. duplication elimination problem that still arises even if the exploration is directed. The class of *memory-restricted search algorithms* has been developed under this aim. The framework imposes an absolute upper bound on the total memory the algorithm may use, regardless of the size of the problem space. If the number of nodes with distance value smaller than the optimal solution path length is larger than this memory bound, storing the entire list of visited nodes is no longer possible.

*Iterative deepening A**, IDA* for short [16], has proven effective to successively search the problem graph with bounded DFS traversals according to an increasing threshold for the tentative values. IDA* consumes space linear in the solution length. It does not use additionally available memory and traverses all generating paths. Unfortunately, the number of paths in a graph might be exponentially larger than the number of nodes such that the design of informative consistent heuristics and duplicate elimination remains essential. If all merits are distinct, IDA* expands a quadratic number

$$
\begin{array}{l}
\textbf{IDA*}(s) \\
\quad\quad \texttt{Push}(S, s, h(s)); U \leftarrow h(s) \\
\quad\quad \textbf{while } (U \neq \infty) \\
\quad\quad\quad U \leftarrow U'; U' \leftarrow \infty \\
\quad\quad\quad \textbf{while } (S \neq \emptyset) \\
\quad\quad\quad\quad (u, f(u)) \leftarrow \texttt{Pop}(S) \\
\quad\quad\quad\quad \textbf{if } (goal(u)) \textbf{ return } (u, f(u)) \\
\quad\quad\quad\quad \textbf{for all } v \textbf{ in } \Gamma(u) \\
\quad\quad\quad\quad\quad\quad \textbf{if } (f(u) + w(u,v) - h(u) + h(v) > U) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{if } (f(u) + w(u,v) - h(u) + h(v) < U') \\
\quad\quad\quad\quad\quad\quad\quad\quad U' \leftarrow f(u) + w(u,v) - h(u) + h(v) \\
\quad\quad\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad\quad\quad \texttt{Push}(S, v, f(u) + w(u,v) - h(u) + h(v))
\end{array}
$$

Table 1. The IDA* Algorihm implemented with a Stack.

of nodes in the worst case. Although iterative deepening is limited to small integral weights it performs well in practice. Table 1 depicts a possible implementation of IDA* in pseudo-code: S is a stack for backtracking, U is the current threshold, and U' the threshold for the next iteration. The value $w(u, v)$ is the weight of the transitition (u, v), $h(u)$ and $f(u)$ is the heuristic estimate and combined merit for node u, respectively.

Pattern data-bases [4] are a general tool to improve the estimate that can cope with complex subproblem interactions. A solution preserving relaxation of the search problem is traversed prior to the search and the goal distances of all abstract states are kept as lower bound estimats for the overall problem within a large hash table. However, the application of this pre-compilation technique is limited to suitable domain abstractions that yield better results than on-line computations as findings in protocol verification [8], AI-planning [6], and selected single-agent problems [14] indicate. Therefore, to lessen memory consumption according to a large number of states is still a problem.

Transposition tables are used to store and improve the distances until the memory bound has been reached [18]. However, when the memory is exhausted, IDA*'s time consumption is often stinged by uncaught duplicates.

Different node caching strategies have been applied: MREC [21] switches from A* to IDA* if the memory limit is reached. In contrast, SMA* [19] reassigns the space by dynamically deleting a previously expanded node, propagating up computed f-values to the parents in order to save re-computation as far as possible. However, the effect of node cashing is still limited. An adversary may request the nodes just deleted.

The paper is aimed to close this gap and is structured as follows: The first section gives an $O(\log n)$-space algorithm to search for the shortest path in graphs with uniform or small weights, with n being the total number of nodes in the problem graph. Suffix lists are a data structure for maximizing the number of stored states according to a given memory limit. The achieved result is compared to ordinary hashing and a derived information-theoretic bound. Bit-state state compaction, sequential hashing and partial search can substitute the transposition table of IDA* with a bit-vector table. Thereby, it is possible to detect more duplicates in the space while increasing the depth of the

```
                                   Path(a, b, l)
Divide-And-Conquer-BFS(s)            if ((a, b) ∈ E)
  for i ← 1 to n                       return true
    for l ← 1 to n                   else
      if (Path(s, i, l))               for j ← 1 to n
        print (s, i, l)                  if (Path(a, j, ⌈l/2⌉) and Path(j, b, ⌊l/2⌋))
        break                              return true
                                       return false
```

Table 2. Computing the BFS Level.

search. We give promising experimental results in validating an industrial communication protocol.

2 Minimum Space Algorithms

First of all, we might ask for the limit of space reduction. Given a graph with n nodes we are interested in algorithms that compute the BFS-level and shortest paths of all nodes and either consume as little working space as possible or perform faster if more space is available. In addition, we assume that the algorithms are not allowed to modify the input during the exection.

The similar problems of node reachability (i.e., determine whether there any path between two nodes) and graph connectivity have been efficiently solved for the same restricted memory setting using random walk strategies [10, 11]. However, we are not aware of any equivalent results for BFS and shortest paths. In the following we will devise an $O(\log n)$ space algorithm for BFS and shortest paths with small integer weights. The principle is similar to the simulation of nondeterministic Turing machines [20].

2.1 Divide-And-Conquer BFS

To compute the breadth-first-level for each node, with very limited space, we may use a DAC strategy *Path* that reports if there is a path from a to b with l edges. If l equals 1 and there is an edge from a to b then the procedure returns true. Otherwise, for each node index j, $1 \leq j \leq n$, we recursively determine *Path*$(a, j, \lceil l/2 \rceil)$ and *Path*$(j, b, \lfloor l/2 \rfloor)$. If both exist the returned value is true, compare Table 2. The recursion stack has to store at most $O(\log n)$ frames each of which contains $O(1)$ integers. Hence the space complexity is $O(\log n)$. However, this has to be paid with a time complexity of $O(n^{3+\log n})$ due to the recurrence equation $T(1) = 1$ and $T(l) = 2n \cdot T(l/2)$ resulting in $T(n) = (2n)^{\log n} = n^{1+\log n}$ time for one test. Varying b and iterating on l in the range of $\{1, \dots, n\}$ gives the overall performance of $O(n^{3+\log n})$ steps.

2.2 Divide-And-Conquer SSSP

To extend this idea to the single-source shortest path problem (cf. Figure 3) with edge weights bounded by a constant C, we check the weights

```
Divide-And-Conquer-SSSP(s)
  for i ← 1 to n
    for w ← 1 to C · n
      if (Path(s, i, w))
        print (s, i, w)
        break

Path(a, b, w)
  if (weight(a, b) = w)
    return true
  else
    for j ← 1 to n
      for s ← max{1, ⌊w/2⌋ − ⌈C/2⌉}
            to min{w − 1, ⌈w/2⌉ + ⌈C/2⌉}
        if (Path(a, j, s) and Path(j, b, w − s))
          return true
    return false
```

Table 3. Searching the Shortest Paths.

$\lfloor w/2 \rfloor - \lceil C/2 \rceil$ for path 1,	$\lfloor w/2 \rfloor + \lceil C/2 \rceil$ for path 2,
$\lfloor w/2 \rfloor - \lceil C/2 + 1 \rceil$ for path 1,	$\lfloor w/2 \rfloor + \lceil C/2 \rceil - 1$ for path 2,
...	...
$\lfloor w/2 \rfloor + \lceil C/2 \rceil$ for path 1,	$\lfloor w/2 \rfloor - \lceil C/2 \rceil$ for path 2.

If there is a path with total weight w then it can be decomposed into one of above partitions. The worst-case reduction on weights is $Cn \to Cn/2 + C/2 \to Cn/4 + 3C/4 \to \ldots \to C \to C-1 \to C-2 \to C-3 \to \ldots \to 1$. Therefore, the recursion depth is bounded by $\log(Cn) + C$ which results in a space requirement of $O(\log n)$ integers. As in the BFS case this compares to exponential time.

We do not claim practical applicability of these algorithms but want to make a start towards efficient shortest path algorithms for relatively little memory and unmodifiable large data, for example on optical read-only storage. In particular, time-space trade-offs seem to require new techniques.

3 Suffix Lists

Given m bits of memory, we want to maintain a dynamically evolving visited list *closed* under inserts and membership queries. The entries of *closed* are integers from $\{0, n\}$. Let r denote the maximal size of closed nodes that can be accommodated. As long as $n \leq m$ a simple bit array with bit i denoting element i is sufficient. Using hashing with open addressing, r is limited to $O(n/\log n)$. In the following we describe a simple but very space efficient approach with small update and query times. Similar ideas appeared in [2] but the data structure there is static and not theoretically analyzed. Another dynamic variant achieving asympotically equivalent storage bounds as our approach is sketched in [1]. However, constants are only given for two static examples. We provide constants for the dynamic version; comparing with the numbers of [1], our dynamic version could host up to five times more elements of the same value range. However, one has to take into consideration that the data structure of [1] provides constant access time whereas our structure incurs amortized logarithmic access time.

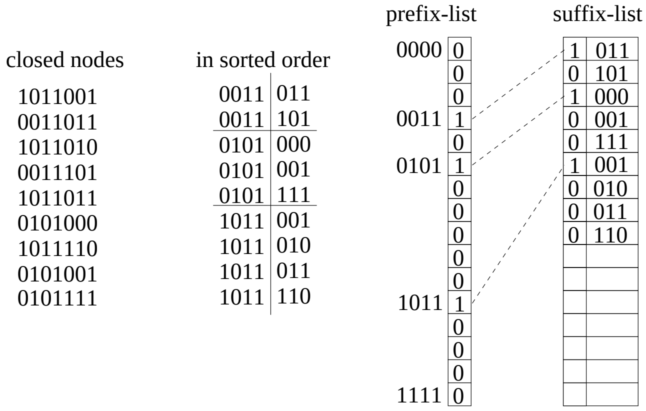

Fig. 1. Example for Suffix Lists with $p = 4$, and $s = 3$.

3.1 Representation

Let *bin*(u) be the binary representation of an element $u \leq n$ from the set *closed*. We split *bin*(u) in p high bits and $s = \lceil \log n \rceil - p$ low bits. Furthermore, $u_{s+p-1}, \ldots, u_s$ denotes the prefix of *bin*(u) and $u_{s-1}, \ldots, u_0$ stands for the suffix of *bin*(u).

A *suffix list data structure* consists of a linear array P of size 2^p bits and of a two-dimensional array L of size $r(m + 1)$ bits. The basic idea of suffix lists is to store a common prefix of several entries as a single bit in P, whereas the distinctive suffixes form a group within L. P is stored as a bit array. L can hold several groups with each group consisting of a multiple of $s + 1$ bits. The first bit of each $s + 1$-bit row in L serves as a *group bit*. The first s bit suffix entry of a group has group bit one, the other elements of the group have group bit zero. We place the elements of a group together in lexicographical order, see Figure 1.

3.2 Searching

First, we compute $k = \sum_{i=0}^{p-1} u_{s+i} \cdot 2^i$ which gives us the search position in the prefix array P. Then we simply count the number of ones in P starting from position $P[0]$ until we reach $P[k]$. Let z be this number. Finally we search through L until we have found the zth suffix of L with group bit one. If we have to perform a membership query we simply search in this group. Note that searching a single entry may require scanning large areas of main memory.

3.3 Inserting

To insert entry u we first search the corresponding group as described above. In case u opens a new group within L this involves setting group bits in P and L. The suffix of u is inserted in its group while maintaining the elements of the group sorted. Note that an insert may need to shift many rows in L in order to create space at the desired position. The maximum number r of elements that can be stored in S bits is limited as follows:

We need 2^p bits for P and $s+1 = \lceil \log n \rceil - p + 1$ bits for each entry of L. Hence, we choose p so that r is maximal subject to

$$r \leq \frac{m - 2^p}{\lceil \log n \rceil - p + 1}.$$

For $p = \Theta(\log m - \log\log(n/m))$ the space requirement for both P and the suffixes in L is small enough to guarantee $r = \Theta\left(\frac{m}{\log(n/m)}\right)$.

3.4 Checkpoints

We now show how to speed up the operations. When searching or inserting an element u we have to compute z in order to find the correct group in L. Instead of scanning potentially large parts of P and L for each single query we maintain checkpoints, *one-counters*, in order to store the number of ones seen so far. Checkpoints are to lie close enough to support rapid search but must not consume more than a small fraction of the main memory. For $2^p \leq r$ we have $z \leq r$ for both arrays, so $\lceil \log r \rceil$ bits are sufficient for each one-counter.

Keeping one-counters after every $1/(c_1 \cdot \lfloor \log r \rfloor)$ entries limits the total space requirement. Binary search on the one-counters of P now reduces the scan-area to compute the correct value of z to $c_1 \cdot \lfloor \log r \rfloor$ bits.

Searching in L is slightly more difficult because groups could extend over 2^s entries, thus potentially spanning several one-counters with equal values. Nevertheless, finding the beginning and the end of large groups is possible within the stated bounds. As we keep the elements within a group sorted, another binary search on the actual entries is sufficient to locate the position in L.

3.5 Buffers

We now turn to insertions where two problems remain: adding a new element to a group may need shifting large amount of data. Also, after each insert the checkpoints must be updated. A simple solution uses a second buffer data structure *BU* which is less space efficient but supports rapid inserts and look-ups. When the number of elements in *BU* exceeds a certain threshold, *BU* is merged with the old suffix lists to obtain a new up-to-date space efficient representation. Choosing an appropriate size of *BU*, amortized analysis shows improved computational bounds for inserts while achieving asymptotically the same order of phases for the graph search algorithm.

Note that membership queries must be extended to *BU* as well. We implement *BU* as an array for hashing with open addressing. *BU* stores at most $c_2 \cdot r / \lceil \log n \rceil$ elements of size $p + s = \lceil \log n \rceil$, for some small constant c_2. As long as there is 10% space left in *BU*, we continue to insert elements into *BU* otherwise *BU* is sorted and the suffixes are moved from *BU* into the proper groups of L. The reason not to exploit the full hash table size is again to bound the expected search and insert time within *BU* to a constant number of tests.

Theorem 1. *Searching and inserting n items into suffix lists under space restriction m can be done in $O(n \cdot \log^2 n)$ bit operations. Assuming $\log n$ bits for a machine word, the total run time for n inserts and memberships is $O(n \log n)$.*

Proof. For a membership query we perform binary searches on numbers of $\lceil \log r \rceil$ bits or s bits, respectively. So, to search an element we need $O(\log^2 r + s^2) = O(\log^2 n)$ bit operations since $r \leq n$ and $s \leq \log n$.

Each of the $O(r/\log n)$ buffer entries consists of $O(\log n)$ bits, hence sorting the buffer can be done with

$$O\left(\log n \cdot \frac{r}{\log n} \cdot \log \frac{r}{\log n}\right) = O(r \cdot \log n)$$

bit operations. Starting with the biggest occurring keys merging can be performed in $O(1)$ memory scans, $O(m)$ operations. This also includes updating all one-counters. In spite of the additional data structures we still have

$$r = \Theta\left(\frac{m}{\log(n/m)}\right).$$

Thus, the total bit complexity for n inserts and membership queries is given by

$$\begin{aligned}
&O(\#\textit{buffer-runs}\ (\#\textit{sorting-ops} + \#\textit{merging-ops}) + \\
&\quad \#\textit{elements}\ \#\textit{buffer-search-ops} + \\
&\quad \#\textit{elements}\ \#\textit{membership-query-ops}) = \\
&O(n/r \cdot \log n \cdot (r \cdot \log n + m) + n \cdot \log^2 n + n \cdot \log^2 n) = \\
&O(n/r \cdot \log n \cdot (r \cdot \log n + r \cdot \log(n/m)) + n \cdot \log^2 n) = \\
&O(n \cdot \log^2 n).
\end{aligned}$$

Assuming a machine word length of $\log n$ in the RAM model, any modification or comparison of entries with $O(\log n)$ bits appearing in our suffix lists can be done using $O(1)$ machine operations. Hence the total complexity reduces to $O(n \cdot \log n)$ operations.

The constants can be improved using the following observation: in the case $n = (1 + \epsilon) \cdot m$, for a small $\epsilon > 0$ nearly half of the entries in P will always be zero, namely those which are lexicographically bigger than the suffix of n itself. Cutting the P array at this position leaves more room for L which in turn enables us to keep more elements.

3.6 The Information Theoretic Bound

We place an upper bound on the maximal size r^* of the subset that can be stored. For the static case, we observe that $\lceil \log \binom{n}{r^*} \rceil \leq m$. However, if we consider the dynamic case, i.e. including insertions, we have to represent all former configurations. This results in

$$\left\lceil \log \left(\sum_{i=0}^{r^*} \binom{n}{i} \right) \right\rceil \leq m.$$

We aim choose r^* maximal subject to this inequality. For $r^* \leq (n-2)/3$ we have

$$\binom{n}{r^*} \leq \sum_{i=0}^{r^*} \binom{n}{i} \leq 2 \cdot \binom{n}{r^*}.$$

The correctness follows from $\binom{n}{i}/\binom{n}{i+1} \leq 1/2$ for $i \leq (n-2)/3$. We are only interested in the logarithms, so we conclude

$$\log \binom{n}{r^*} \leq \log \left(\sum_{i=0}^{r^*} \binom{n}{i} \right) \leq \log \left(2 \binom{n}{r^*} \right) = \log \binom{n}{r^*} + 1$$

Obviously in this restricted range it is sufficient to concentrate on the last binomial coefficient. The error in our estimate is at most one bit. The restriction on r^* is compatible with all reasonable choices for n and m. Using

$$\begin{aligned} \log \binom{n}{r^*} &= \log \frac{n \cdot (n-1) \cdot \ \cdots \ \cdot (n - r^* + 1)}{r^*!} \\ &= \sum_{j=n-r^*+1}^{n} \log j - \sum_{j=1}^{r^*} \log j, \end{aligned}$$

we can approximate the logarithm by two corresponding integrals. If we properly bias the integral limits we can be sure to compute a lower bound

$$\log \binom{n}{r^*} \geq \int_{n-r^*+1}^{n} \log(x)\, \mathrm{d}x - \int_{2}^{r^*+1} \log(x)\, \mathrm{d}x.$$

Maximizing r^* with respect to this equation yields an information theoretic upper bound.

Table 4 compares suffix lists with hashing and open addressing. The constants for suffix lists are chosen so that $2 \cdot c_1 + c_2 \leq 1/10$ which means that if r elements can be treated, we set aside $r/10$ bits to speed-up internal computations. For hashing with open addressing we also leave 10% memory free to keep the internal computation time moderate. When using suffix lists instead of hashing, note that only the ratio between n and m is important. For the static data structure of [1] the following numbers are given: for $\frac{n}{m} = \frac{1.0 \cdot 2^{32}}{1.9 \cdot 2^{30}} \approx 1.05$ it can store a fraction of $\frac{r}{n} = \frac{1.4 \cdot 2^{27}}{1.0 \cdot 2^{32}} \approx 4.37\%$ of n. Our approach achieves 22.7% which constitues an improvement by a factor of more than five. For another example with $n/m \approx 3.2$ our approach gains by a factor of about 1.8.

Hence, suffix lists can close the phase gap in search algorithms between the upper bound and trivial approaches like hashing with open addressing. Already for $n \geq 1.1 \cdot m$ we reach two-optimality.

4 Bit-State Hash-Tables

Advanced to the treatment of data structures and algorithms we give a small introduction to the verification of distributed software systems and communication protocols; an apparent and practical relevant domain for state-space search.

n/m	Upper Bound	Suffix Lists	Hashing $n = 2^{20}$	Hashing $n = 2^{30}$
1.05	33.2 %	22.7 %	4.3 %	2.9 %
1.10	32.4 %	21.2 %	4.1 %	2.8 %
1.25	24.3 %	17.7 %	3.6 %	2.4 %
1.50	17.4 %	13.4 %	3.0 %	2.0 %
2.00	11.0 %	9.1 %	2.3 %	1.5 %
3.00	6.1 %	5.3 %	1.5 %	1.0 %
4.00	4.1 %	3.7 %	1.1 %	0.7 %
8.00	1.7 %	1.5 %	0.5 %	0.4 %
16.00	0.7 %	0.7 %	0.3 %	0.2 %

Table 4. Fractions of n stored in Suffix Lists and Hashing with Open Addressing.

4.1 State Space Search for Protocols Validation

Reliable communication is probably the most important issue for accessing the Internet and for the design of distributed computer systems. Usually a layered structure like the ISO Reference Model is used to allow for different abstractions. In one layer (transport layer) we have the request for reliable communication while the next lower layers provide this quality of service facing a lossy channel (cf. Figure 2).

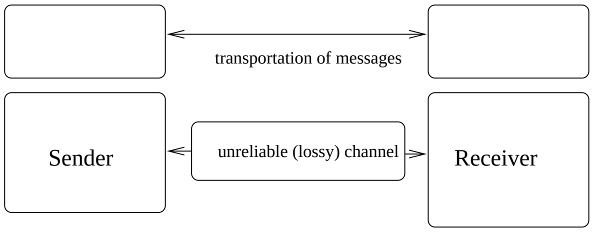

Fig. 2. Communication over a Lossy Channel for Messaging in Layered Protocols.

One example to cope with lossy channels is the alternating bit protocol. The message flow is visualized in Figure 3. To assert secure data transport from the sender to the receiver we assume sequence numbers for messages. In the following we study algorithms and data structures to certify the correctness of a such a protocol.

4.2 Supertrace

The idea of bit-state hashing is adopted from Holzman's protocol validator Spin [12], that parses the expressive concurrent Promela protocol specification language. It compresses the state description of several hundred bits down to only a few bits to build a hash table with up to 2^{30} entries and more. Combined with a depth-first search strategy this is in fact the *supertrace algorithm*: A state s is represented by its hash address $h(s)$. When generating a state the corresponding bit is set. Synonyms are regarded as duplicates resulting in pruning the search. The search algorithm is not complete, since not all synonyms are disambiguated. Moreover, through depth-first traversal, the length of a witness for an encountered error state is not minimal.

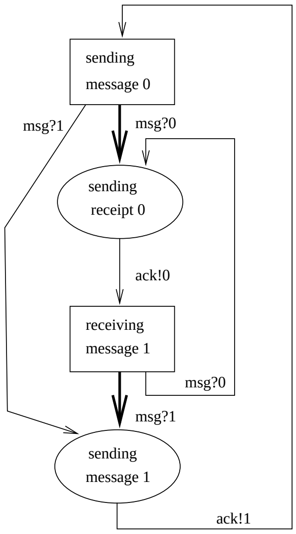

Fig. 3. Flow of Control on a Lossy Channel with the Alternating Bit Protocol.

4.3 Data Structures

As an illustration and generalization of the bit-state hashing idea, Figure 4 depicts the range of possible hash structures: Usual hashing with chaining of synonyms, single-bit hashing, double-bit hashing and hash compact [22]. Let n be the number of reachable states and m be the maximal number of bits available. A coarse approximation for single bit-state hashing coverage with $n < m$ is $1 - P_1$ with the average probability of collision $P_1 \leq \frac{1}{n}\sum_{i=0}^{n-1} \frac{i}{m} \leq n/2m$, since the i-th element collides with one of the $i-1$ already inserted elements with a probability of at most $(i-1)/m$, $1 \leq i \leq n$ [13]. For multi-bit hashing and h (independent) hash-functions by assuming $hn < m$ coverage is improved to $1 - P_h$ with average probability of collision $P_h \leq \frac{1}{n}\sum_{i=0}^{n-1}(h \cdot \frac{i}{m})^h$, since i elements occupy at most hi/m addresses, $0 \leq i \leq n-1$. For double bit-state hashing this simplifies to $P_2 \leq \frac{1}{n}(\frac{2}{m})^2 \sum_{i=0}^{n-1} i^2 = 2(n-1)(2n-1)/3m^2 \leq 4n^2/3m^2$.

4.4 Sequential and Universal Hashing

The drawback in incompleteness of partial search is compensated by re-invoking the algorithm with different hash functions to improve the coverage of the search tree. Subsequently, this technique, called *sequential hashing*, examines various beams in the search tree (up to a certain threshold depth). In considerably large protocols supertrace with sequential hashing succeeds in finding bugs but still returns long witness trails. If in sequential hashing exploration with the first hash first function covers m/n of the search space, the probability that a state x is not generated in d independent runs is $(1 - m/n)^d$ such that x is reached with probability $1 - (1 - m/n)^d$. Eckerle and Lais [5] have shown that this *ideal* circumstances are not given in practice and refine the model for coverage prediction.

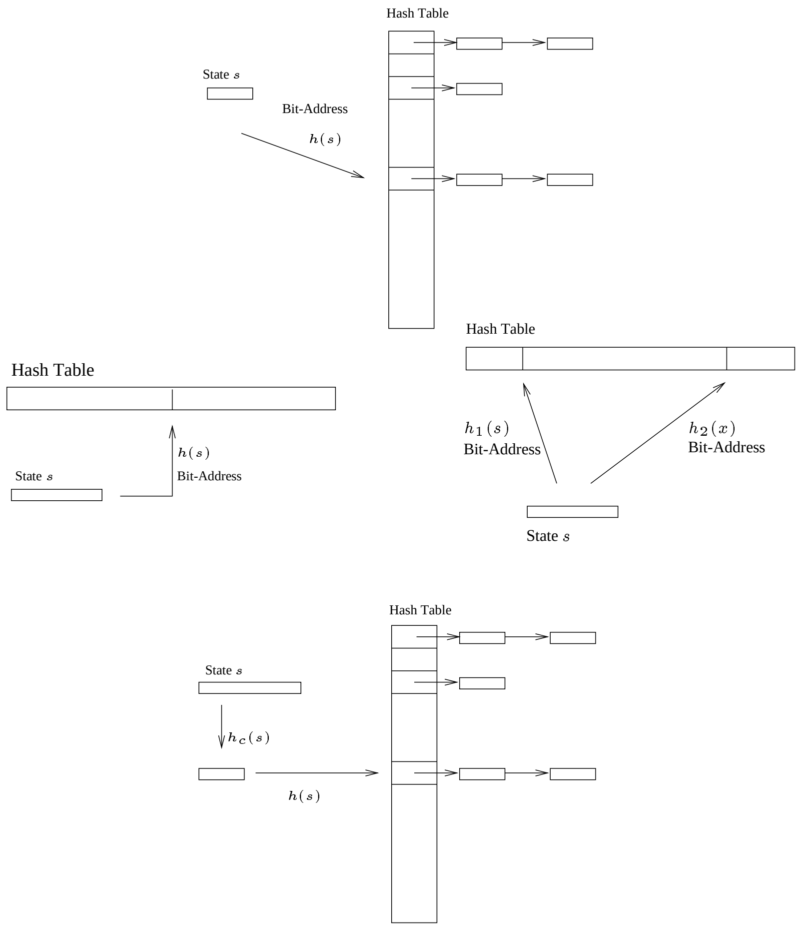

Fig. 4. Ordinary Hashing, Single Bit-State Hashing, Double Bit-State Hashing, and Hash-Compact.

Moreover, universal hash functions suit best for implementing sequential hashing. Let A, B be sets with $|B| = 2^w$, for some integer value w. The class of hash functions $\mathcal{H}$ is *universal*, if for all $x, y \in A$, we have

$$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{|B|}.$$

Universal hash functions lead to a good distribution of values on the average. If h is drawn randomly from $\mathcal{H}$ and S is the set of keys to be inserted in the hash table, the expected cost of each search, insert and delete operation is bounded by $(1 + |S|/|B|)$. We give an example of a universal hash function. Let p be prime, and $p \geq |A|$ and $h_{m,n}(s) = ((m \cdot s + n) \bmod p) \bmod |B|$. Then the class $\mathcal{H}_1 := \{h_{m,n} | m, n \in Z_p\}$ is universal.

```
Partial IDA*(s)
    Push(S, s, h(s)); U' ← U ← h(s)
    while (U ≠ ∞)
        U ← U'; U' ← ∞
        Init(H)
        while (S ≠ ∅)
            (u, f(u)) ← Pop(S)
            if (goal(u)) return (u, f(u))
            for all v in Γ(u)
                if (Search(H, v) ≠ ∅)
                    Insert(H, v)
                    if (f(u) + w(u, v) − h(u) + h(v) > U)
                        if (f(u) + w(u, v) − h(u) + h(v) < U')
                            U' ← f(u) + w(u, v) − h(u) + h(v)
                    else
                        Push(S, v, f(u) + w(u, v) − h(u) + h(v))
```

Table 5. Partial IDA* Algorithm.

4.5 Validating Process

For the validation of the design of the protocols, bug-finding by simulation and testing has its drawbacks, since several subtle bugs in concurrent systems are difficult to establish. Given a formal specification of a desired protocol property model-checking is a push-button procedure to verify the correctness. Validation is performed by traversing the finite-state machine representation of the protocol to find a bug. Therefore protocols are represented by state spaces, in which reachability analysis is performed to establish error states.

Therefore, directed search for minimal counterexamples in the protocol space according to a given implementation corresponds to the search for an optimal solution with the goal as the failure state. From a model checking perspective [3] the approach allows to implement various heuristics to direct the search into the direction of the failure. From an AI-perspective partial search, maybe assisted with sequential hashing, condenses duplicate information in various search and planning problem spaces.

4.6 Heuristic Search Algorithm

The apparent aspirant for state compaction is IDA* with *transposition tables*, since, in opposite to A*, it tracks the solution path on the stack, which allows to omit the predecessor link in the state description of the set of visited states.

When substituting the transposition table H of already visited nodes in IDA* by bit-state, multi bit-state or hash compaction we establish the Partial IDA* algorithm as depicted in Table 5. Since neither the predecessor nor the f-value are present, in order to distinguish the current iteration from the previous ones, the bit-state table has to be re-initialized in each iteration of IDA*. Refreshing large bit-vector tables is fast in practice, but for shallow searches with a small number of expanded nodes this scheme can be improved by invoking ordinary IDA* with transposition table updates for smaller thresholds and by applying bit-vector exploration in large depths only.

In practice the obtained counterexamples are minimal, since the coverage with bit-state duplicate elimination is very close to 100 % for moderately sized systems ($n < m$). Moreover, the technique of *trail-directed search* can effectivly improve non-optimal existing paths [9].

The results for searching deadlocks in one large communication protocol are depicted in Table 6, where the number of expansions with respect to different optimal search algorithms for an increasing threshold is shown. For A* a snapshot is taken at each time the priority queue value increases, while in IDA* the number of expanded nodes according to each completed iteration is shown. Hence, the number of node expansion numbers in these two algorithms do not match exactly, but indicate a common trend. The considered protocol instance is the industrial General Inter-ORB Protocol (GIOP, 1 server and 3 clients) [15], which is a key component of the Common Object Request Broker Architecture (CORBA) specification.

The witness for a seeded deadlock in depth 70 has to be established according to the heuristic that counts the number of non-active processes. The state vector generated by the validator tool SPIN is 544 Bytes large, such that the visited list (hash table or transposition table) is bounded to 2^{18} states corresponding to approx. 2^{17} KByte or 128 MByte. Therefore, we fix the size of the bit-state hash table accordingly at 2^{30} Bits.

Algorithm A* exceeds its space limit in depth 61 and fails. IDA* utilizes a transposition table which is exhausted at the same depth. As IDA* then searches the tree of generation paths it compensates space for time. But even when investing more than 24 hours on our 248 MHz Sun Ultra Workstation and when utilizing the table constructed so far, ordinary IDA* was not able to complete search depth 61. On the other hand, Partial A* finishes all searches up to depth 70 with either single- and double bit-state hashing within a total of one hour.

Since the algorithms are not complete, we validated optimality with A* with our maximum of 1.5 GByte main memory. Note that the difference in the number of node expansions in single and double bit-state hashing is very small (less than a hundred) and only occurs in large search depths (iteration 58 onwards). As Partial IDA* with double bit-state hashing expands exactly the same number of states as IDA* with a transposition table, we actually observe no loss of information in the example.

5 Conclusion

At the limit of main memory eliminating duplicates and weight diversity can soon result in thrashing both resources time and space, such that powerful data structures for caching, partial search and compressed dictionaries are required. Therefore, regarding the limits and possibilities of A*, we have suggested different contributions to memory-restricted search. Partial search supports bookkeeping in tremendously large hash tables to avoid duplicates in the search, while suffix lists push the envelope for increasing the number of nodes to be stored without loss of information.

The treatment of Partial IDA* search elaborates on precursoring findings in [8], where a rudimentory bit vector and single-bit hashing function has been chosen for im-

depth	A* (hash table)	IDA* (transposition table)	Partial IDA* (single bit-state)	Partial IDA* (double bit-state)
⋮	⋮	⋮	⋮	⋮
40	6,646	6,333	6,333	6,333
41	9,306	8,184	8,184	8,184
42	10,955	10,575	10,575	10,575
43	13,666	13,290	13,290	13,290
44	17,761	16,500	16,500	16,500
45	20,130	19,860	19,860	19,860
46	25,426	23,646	23,646	23,646
47	27,714	27,654	27,654	27,654
48	33,799	32,040	32,040	32,040
49	37,095	37,011	37,011	37,011
50	46,105	42,849	42,849	42,849
51	51,113	49,872	49,872	49,872
52	61,710	58,545	58,545	58,545
53	73,195	69,162	69,162	69,162
54	85,245	81,993	81,993	81,993
55	96,995	96,543	96,543	96,543
56	113,950	112,296	112,296	112,296
57	115,460	129,138	129,138	129,138
58	147,042	146,625	146,623	146,625
59	150,344	164,982	164,978	164,982
60	184,872	184,383	184,376	184,383
61	187,411	206,145	206,135	206,145
62	-	$>$ 97,157,721	229,611	229,626
63	-	-	255,386	255,411
64	-	-	282,416	282,444
65	-	-	311,306	311,340
66	-	-	341,522	341,562
67	-	-	373,374	373,422
68	-	-	407,249	407,310
69	-	-	442,863	442,941
70	-	-	67	67

Table 6. Number of Expanded nodes of Search Algorithms in the GIOP Protocol.

plementation. For the experiments we chose a non-trivial protocol example [7], but recent progress shows that the algorithm has also reduced the search efforts for optimally solving Atomix, a PSPACE-complete AI single-agent search problem [14]. Omitting the visited list and exploring the space in a Divide-and-Conquer fashion has been proposed in [17], and the algorithms we consider study the effect of removing the horizon-list as well. Another model checking approach for state compression as to answer to the representation problem of large sets of states are binary decision diagrams (BDDs) that are able to encode large sets of states without necessarily encountering exponential growth. However, hybrid methods of explicite and symbolic search methods are still to be developed.

Acknowledgements The authors would like to thank Robert Holte for valuable discussions and comments and Alberto Lluch Lafuente for the joined implementation efforts in the HSF-SPIN protocol validator. The work is partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by DFG within the project *Heuristic Search and Its Application to Protocol Validation.*

References

1. A. Brodnik and J. Munro. Membership in constant time and almost-minimum space. *SIAM*, 28(3):1627–1640, 1999.
2. Y. Choueka, A. Fraenkel, S. Klein, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. pages 88–96, 1986.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
4. J. C. Culberson and J. Schaeffer. Searching with pattern databases. Lecture Notes in Computer Science, pages 402–416. Springer, 1996.
5. J. Eckerle and T. Lais. Limits and possibilities of sequential hashing with supertrace. In *IFIP FORTE/PSTV*, Lecture Notes in Computer Science. Springer, 1998.
6. S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, 2001. To appear.
7. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *SPIN Workshop*, Lecture Notes in Computer Science, pages 57–79. Springer, 2001.
8. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.
9. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Trail-directed model checking. In *Workshop on Software Model Checking*. Electronic Notes in Theoretical Computer Science, Elsevier, 2001. To appear.
10. U. Feige. A fast randomized LOGSPACE algorithm for graph connectivity. *Theoretical Computer Science*, 169(2):147–160, 1996.
11. U. Feige. A spectrum of time-space tradeoffs for undirected $s - t$ connectivity. *Journal of Computer and System Sciences*, 54(2):305–316, 1997.
12. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
13. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design: An International Journal*, 13(3):289–307, 1998.
14. F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to Atomix. This volume.
15. M. Kamel and S. Leue. Formalization and validation of the general inter-orb protocol (GIOP) using Promela and SPIN. In *Software Tools for Technology Transfer*, volume 2, pages 394–409, 2000.
16. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
17. R. E. Korf. Divide-and-conquer bidirectional search: First results. In *IJCAI*, pages 1184–1189, 1999.
18. A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.
19. S. Russell. Efficient memory-bounded search methods. In *ECAI-92*, pages 1–5, 1992.
20. W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.
21. A. K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, pages 297–302, 1989.
22. U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90. Shaker Verlag, Aachen, 1996.