Succinct Set-Encoding for State-Space Search

Tim Schmidt and **Rong Zhou**
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

We introduce the level-ordered edge sequence (LOES), a succinct encoding for state-sets based on prefix-trees. For use in state-space search, we give algorithms for member testing and element hashing with runtime dependent only on state size, as well as time and memory efficient construction of and iteration over such sets. Finally we compare LOES to binary decision diagrams (BDDs) and explicitly packed set-representation over a range of IPC planning problems. Our results show LOES produces succinct set-encodings for a wider range of planning problems than both BDDs and explicit state representation, increasing the number of problems that can be solved cost-optimally.

Introduction

A key challenge of state-space search is to make efficient use of available memory. Best-first search algorithms such as A* are often constrained by the amount of memory used to represent state-sets in the search frontier (the Open list), previously expanded states (the Closed list), and memory-based heuristics including pattern databases (Culberson and Schaeffer 1998).

Although linear-space search algorithms such as IDA* (Korf 1985) solve the memory problem for A*, they need extra node expansions to find an optimal solution, because linear-space search typically does not check for duplicates when successors are generated and this can lead to redundant work. Essentially, these algorithms trade time for space and their efficiency can vary dramatically, depending on the number of duplicates encountered in the search space. For domains with few duplicates such as the sliding-tile puzzles, IDA* easily outperforms A*. But for other domains such as multiple sequence alignment, linear-space search simply takes too long. For STRIPS planning, IDA* can expand many more nodes than A*, even if it uses a transposition table (Zhou and Hansen 2004). As a result, state-of-the-art heuristic search planners such as Fast Downward (Helmert 2006) use A* instead of IDA* as their underlying search algorithm.

A classic approach to improving the storage efficiency of A* is to use BDDs (Bryant 1986) to represent a set (or sets) of states compactly (Jensen, Bryant, and Veloso 2002). By merging isomorphic sub-graphs, BDDs can be much more succinct than an equivalent explicit-state representation. For example, BDD-based planners such as MIPS (Edelkamp and Helmert 2001) perform extremely well on domains like gripper where BDDs are exponentially more compact than explicit-state storage (Edelkamp and Kissmann 2008). However, such showcase domains for BDDs are only a small fraction of the benchmark problems found in planning, and most heuristic search planners still use explicit-state representation, even though they can *all* benefit from having a succinct storage for state-sets.

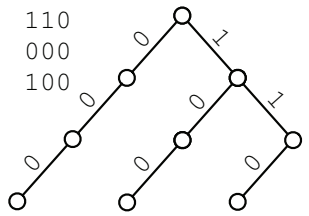

Figure 1: Three bit sequences and their induced prefix tree.

Preliminaries

We assume that the encoding size of a search problem can be determined upfront (e.g., before the start of the search). Without loss of generality, let's assume m is the number of bits required to encode any state for a given problem. Then any set of such states can be represented as an edge-labeled binary tree of depth m with labels $false$ and $true$. Every path from the root to a leaf in such a tree corresponds to a unique state within the set and can be reconstructed by the sequence of edge-labels from the root to the leaf. In the context of this work, we refer to these trees as prefix trees. All elements represented by a subtree rooted by some inner node share a common prefix in their representation denoted by the path from the root to that node. An example is given in Figure 1.

Level-Ordered Edge Sequence

Permuting the state representation

The storage efficiency of a prefix-tree representation depends on the average length of common prefixes shared be-

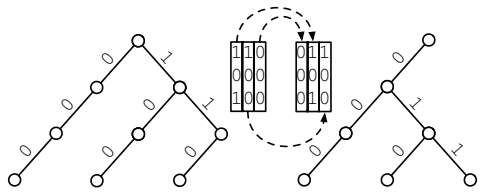

Figure 2: Permuting the encoding's bit-order can reduce the size of a prefix tree.

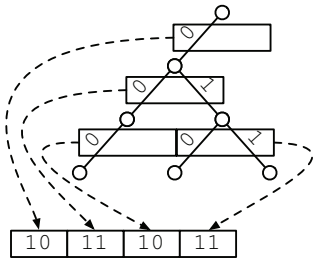

Figure 3: Level-ordered edge sequence for the example set.

tween members of the represented set, which in turn depends on the ordering of bits used in the state-encoding (see Fig.2 for an example). To efficiently find a suitable order, we greedily search through permutations on the bit-order to find one that maximizes the average length of the common prefixes. As a first step we generate a sample set of states starting from a singleton set comprising of the initial state. At each iteration we randomly pick a state from this set, generate its successors and add them back to the set. The process ends after the set has either grown to a specified size or a fixed number of iterations have been executed, whichever happens first. The random selection is aimed at generating a good sample of valid states at different search depths.

Algorithm 1: minimal entropy bit-order

Input: sampleset a set of sampled states
Output: bitorder a permutation of the state encoding

```
subtrees ← {sampleset};
bitorder ← ⟨⟩;
while unassigned bit-positions do
    foreach unassigned bit-position p do
        subtrees^p ← {∅};
        foreach S ∈ subtrees do
            S^p_true ← {s ∈ S : s[p] = true};
            S^p_false ← S/S^p_true;
            subtrees^p ← subtrees^p ∪ {S^p_true} ∪ {S^p_false};
    p* ← argmin_p {H(subtrees^p)};
    bitorder ← bitorder ∘ p*;
    subtrees ← subtrees^p* ;
```

Then we deduce a suitable bit-order by greedily constructing a prefix tree over these sample states in a top-down fashion (see Algorithm 1). Each iteration begins with sets for each leaf node of the current tree, holding the subset with the prefix corresponding to the path from the root to the leaf node. The process starts with a single leaf-set comprising of all sample states, an empty bit-order and all bit-positions designated as candidates. During an iteration we look at each remaining unassigned candidate bit and create a temporary new tree layer by partitioning each set according to the value of this bit in its states. To maximize average prefix lengths we choose the candidate with the least entropy in its leaf-sets as next in the bit-order. It ends after m iterations, when all candidates have been assigned. Intuitively this process will move bits whose value is near-constant in the sample set to the most significant bit positions in the permuted string.

LOES encoding

While the tree representation eliminates prefix redundancy among set-members, straightforward implementations can easily exceed a simple concatenation of the members' bit-strings in size. The culprit are pointers, each of which can take up to 8 bytes of storage on current machines. An alternative are pointer-less structures such as the Ahnentafel[1] representation of binary heaps. A historical Ahnentafel represented the generation order of individuals solely through their positions in the document. At the first position is the subject. The ordering rule is that for any individual at position i, the male ancestor can be found at position $2i$ and the female ancestor at position $2i + 1$ with the offspring to be found at position $\lfloor i/2 \rfloor$. More generally a binary tree is stored in an array through a bijection which maps its elements (individuals) to positions in the array in level-order (i.e. the order of their generation). This technique is well suited for full binary trees (such as binary heaps), but relatively costly for general binary trees. However it can be adapted as we will see in the following.

First, an information-theoretical view on the encoding of binary trees. The number of different binary trees w.r.t. their number of nodes is given by the Catalan numbers.

$C_0 = 1, C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$ for $n \geq 0$

Stirling's approximation gives $\log_2 C_n$ as $2n + O(n)$, which gives us an idea of the information theoretic minimum number of bits required to represent a general binary tree of n nodes. Different encodings exist (see (Jacobson 1988) and (Munro and Raman 2001) for examples) that can store such trees with 2 bits per tree-node and support for basic tree navigation with little additional space overhead. Our prefix-trees are constrained in that all leaves are at depth $m - 1$. We devised an encoding that exploits this property, which results in shorter codes as well as simpler and faster algorithms. We call this encoding *Level-Ordered Edge Sequence* or LOES. It is defined as the level-order concatenation of 2-bit-edge-pair records for each inner node of the tree (the bits corresponding to the presence of the $false$ and $true$ edges at that node). Figure 3 shows how this allows us to encode our

[1] German for ancestor table.

example set in a single byte, little more than half the length of the more general encodings mentioned above.

Space requirements of LOES-sets For a set of n unique states, the prefix tree is maximal, if the average common prefix length of the states is minimal. Intuitively this results in a structure that resembles a perfect binary tree up to depth $k = \lfloor \log_2 n \rfloor$ and degenerate trees from each of the nodes at depth k. Hence the set-tree will at most encompass $2n + n(m - k)$ nodes. For large sets of long states (i.e. $\log_2 n \ll m \ll n$) this is less than $(2 + m)n \approx nm$ nodes. As each node (with the exception of the tree root) has exactly one (incoming) edge and each record in LOES holds at least one edge, the code will at most be little more than twice the length of the concatenation of packed states in the set.

The best case results from the opposite situation, when the structure represents a degenerate tree up to depth $j = m - \lfloor \log_2 n \rfloor$, followed by a perfect binary tree on the lower levels. Such a tree comprises of $2n + (m - j)$ nodes, with each record in the binary tree representing two edges. For large sets (i.e. $m \ll n$), $2n$ bits is hence a tight lower bound on the minimal length of the LOES code.

Tree operations

For use in state-space search or planning we need to enable three different operations in a time and space efficient way: (1) set-member queries, (2) a bijective mapping of a set's n elements to integers $0 \ldots n-1$ to allow efficient association of ancillary data to states and (3) iterating over set elements. All of these operations require efficient navigation through the LOES. For any edge in the sequence at some offset o, the entries for the $false$ and $true$ edges of the node it points to can be found at offsets $2rank(o)$ and $2rank(o) + 1$, where $rank(o)$ is a function that gives the number of set bits in the sequence up to (and including) offset o. This is, because each set bit (present edge) in the LOES code results in an edge-pair record for the target node on the next level (with the exception of the leaf level). As these records are stored in level order, all preceding (in the LOES) edges result in preceding child records. Hence the child record for some edge at offset o will be the $rank(o) + 1$-*th* record in the sequence (as the root node has no incoming edge). Transforming this to offsets with 2-bit records, $2rank(o)$ and $2rank(o)+1$ then give the respective offsets of the presence bits of the target node's $false$ and $true$ edges.

Rank Our implementation makes use of a two-level index, which logically divides the LOES into blocks of 2^{16} bits and sub-blocks of 512 bits. For each block, the index holds an 8-byte unsigned integer, denoting the number of set bits from the beginning of the sequence up to the beginning of the block. On the sub-block level, a 2-byte unsigned value stores the number of set bits from the beginning of the corresponding block up to the beginning of the sub-block. The index overhead within a block is hence

$$\underbrace{\frac{64}{2^{16}}}_{\text{block index}} + \underbrace{\frac{16}{2^{9}}}_{\text{sub-block index}} \approx 0.0323$$

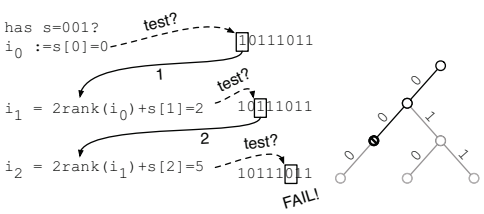

Figure 4: Path offset computation for `001` in the example set. At each level, we compute the offset of the corresponding edge-presence bit, test the bit, continue on pass and return $\perp$ on a fail.

With these indices, the rank function comprises of straightforward look-ups of the block and sub-block indices and set-bit counting within the sub-block.

Algorithm 2: path-offset

Input: state a bitsequence of length m
Output: offset an offset into LOES
Data: LOES an encoded state-set

```
offset ← 0;
for depth ← 0 to m − 1 do
    if state[depth] then
        offset ← offset + 1;
    if LOES[offset] then
        if depth = m − 1 then
            return offset;
        else
            offset ← 2rank_LOES(offset);
    else
        return ⊥;
```

Path-offset The path-offset function (Algorithm 2) navigates through the LOES according to the path interpretation of a state. If the state represents a valid path from the tree root to some leaf, the function returns the offset of the bit corresponding to the last edge of the path. Else it evaluates to $\perp$. An example is given in Figure 4

Member test Based on the offset function, member tests are straightforward. A set contains a state, if and only if its path interpretation corresponds to a valid path through the prefix tree.

Member index Function member index (Algorithm 3) maps states to values $\{\perp, 0, \ldots, n-1\}$. It is bijective for all member states of the set and hence allows associating ancillary data for each member without requiring pointers. The idea is that each set-member's path corresponds to a unique edge at the leaf-level of the tree. The path-offset function gives the address of that bit. By computing the rank of the address, each state is assigned a unique integer in a consecutive range. We normalize these values to the interval $[0; n)$ by subtracting the rank of the last offset of the last but one layer +1. Figure 5 shows this for our example set.

Algorithm 3: member index

Input: state a bitsequence of length m
Data: LOES an encoded state-set
Data: levelOffsets array of offsets

$o \leftarrow$ `path-offset(`state`)`;
if $o = \perp$ **then**
  return $\perp$;
$a \leftarrow \texttt{rank}_{\mathsf{LOES}}(o)$;
$b \leftarrow \texttt{rank}_{\mathsf{LOES}}(\mathsf{levelOffsets}[m-1]-1)$;
return $a - b - 1$;

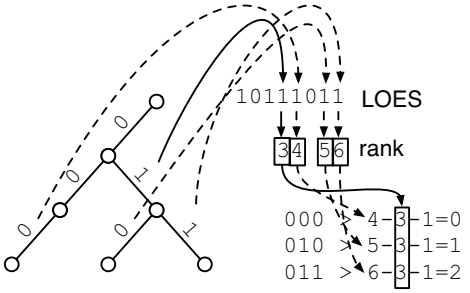

Figure 5: Index mappings for all states in the example set. We subtract the rank+1 (of the offset) of the last edge in the last-but-one level from the rank of the path-offset of an element to compute its index.

Set iteration Set-iteration works by a parallel sweep over the LOES subranges representing the distinct levels of the tree. The first element is represented by the first set bit on each level. The iteration ends after we increased the leaf-offset past the last set bit in the LOES. Algorithm 4 gives the pseudocode for advancing the iteration state from one element to the next element in the set. Conceptually, starting from the leaf-level, we increase the corresponding offset until it addresses a set-bit position. If this advanced the offset past a record boundary (every 2 bits) we continue with the process on the next higher level. Figure 6 shows the different iteration states for our running example. Even-offsets into the LOES correspond to $false$ labels and odd-offsets to $true$ labels. Hence elements can be easily reconstructed

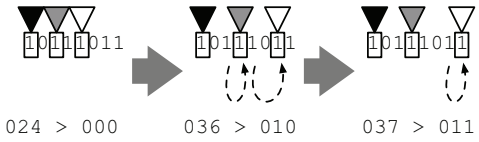

Figure 6: Iteration over the example set. The first row shows the LOES offsets at each iteration state (black, gray and white corresponding to tree levels 0,1,2). The second row shows how these offsets $(\mathrm{mod}\ 2)$ give the elements represented by the LOES.

Algorithm 4: advance iterator

Data: LOES an encoded state-set
Data: offsets an array of LOES offsets

level $\leftarrow m - 1$;
continue $\leftarrow$ true;
while continue & level ≥ 0 **do**
  $\mathsf{rec}_{id} \leftarrow \lfloor \frac{\mathsf{offsets[level]}}{2} \rfloor$;
  repeat
    offsets[level] $\leftarrow$ offsets[level] $+ 1$;
  until LOES[offsets[level]] $= true$;
  continue $\leftarrow \mathsf{rec}_{id} \neq \lfloor \frac{\mathsf{offsets[level]}}{2} \rfloor$;
  level $\leftarrow$ level $- 1$;

from the offsets-array. Note that the iteration always returns elements in lexicographical order. We will make use of this property during set construction.

Set construction

Algorithm 5: add state

Input: s a bitsequence of length m
Data: treelevels an array of bitsequences
Data: s' a bitsequence of length m or $\perp$

if s' $= \perp$ **then**
  depth $\leftarrow -1$;
if s' $=$ s **then**
  return;
else
  depth $\leftarrow i : \forall j < i, \mathsf{s}[j] = \mathsf{s'}[j] \wedge \mathsf{s}[i] \neq \mathsf{s'}[i]$;
  treelevels[depth].lastBit $\leftarrow$ true;
for $i \leftarrow$ depth $+ 1$ **to** $m - 1$ **do**
  if s $[i]$ **then**
    treelevels$[i] \leftarrow$ treelevels$[i] \circ \langle 01 \rangle$;
  else
    treelevels$[i] \leftarrow$ treelevels$[i] \circ \langle 10 \rangle$;
s' $\leftarrow$ s;

LOES is a static structure. Addition of an element in general necessitates changes to the bit-string that are not locally confined, and the cost of a naive insertion is hence $O(n)$. To elaborate on this topic, we first consider how a sequence of lexicographically ordered states can be transformed into LOES. We first initialize empty bit-sequences for each layer of the tree. Algorithm 5 shows how we then manipulate these sequences when adding a new state. If the set is empty, we merely append the corresponding records on all levels. Else, we determine the position or depth d of the first differing bit between s and s', set the last bit of sequence d to $true$ and then append records according to s to all lower levels. Duplicates (i.e. $s = s'$) are simply ignored. After the last state has been added, all sequences are concatenated in level-order to form the LOES. Figure 7 shows this process for our running example. Due to its static nature, it is preferable to add states in batches. An iteration over a

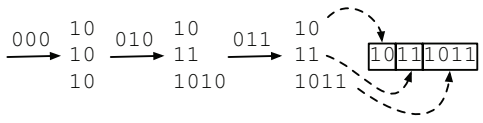

Figure 7: Construction of the set by adding the example states in lexicographic order with algorithm 5. After all states are added, the sequences are concatenated in level-order to form the LOES.

LOES-set returns elements in lexicographical order. In this way, we can iterate over the initial set and the lexicographically ordered batch of states in parallel and feed them to a new LOES in the required order. If the batch is roughly the size of the set, the amortized cost of an insertion is then $O(\log n)$. The $\log$ stems from the need to have the batch in lexicographical order. In this context, we just want to note that any search or planning system based on LOES should be engineered around this property.

Peak-memory requirements To minimize the memory requirements of these merge operations, our bit-sequences comprise of equal size memory chunks. Other than the usual bitwise operators, they support destructive reads in which readers explicitly specify their subrange(s) of interest. This allows to free unneeded chunks early and significantly reduces the peak memory requirements of LOES merges.

Empirical Evaluation

For our empirical evaluation, we concentrate on peak-memory requirements during blind searches in a range of IPC domains. To this end, we implemented a breadth-first search environment for comparative evaluation of LOES and BDDs. We also compared LOES with a state-of-the-art planner, Fast Downward (FD) in its blind-heuristic mode. We used SAS+ representations generated by FD's preprocessor as input for all our tests. The BDD version of our planner is based on the BuDDy package (Lind-Nielsen et al. 2001), which we treated as a black-box set representation. After each layer, we initiated a variable-reordering using the package's recommended heuristic. As the order of expansions within a layer generally differs due to the inherent iteration order of the BDD and LOES versions, we looked at Open and Closed after expanding the last non-goal layer for our tests. During the evaluation, we used 128-byte chunks for LOES' bit-strings, as well as a one megabyte buffer for packed states, before we transformed them into LOES. The test machine used has two Intel 2.26 GHz Xeon processors (each with 4 cores) and 8GB of RAM. No multi-threading was used in the experiments.

Table 1 gives an overview of the results. For the size comparison, the table gives the *idealized* (no padding or other overhead) concatenation of packed states (Packed) as a reference for explicit state storage. In all but the smallest instances, LOES' *actual* peak-memory requirement is well below that of (ideally) Packed. Set-elements on average required between $6 - 58\%$ of the memory of the ideally packed representation on the larger instances of all test domains. As LOES eliminates a significant fraction of the redundancies BDD exploits, its compression rate is analogous to the latter, albeit the variance is much smaller. LOES in particular avoids the blow-up BDDs suffer in domains like freecell, microban and the n-puzzles, showing robustness across all test domains. LOES also does not rely on large sets for good compression, making it even more attractive

Instance	$\|O \cup C\|$	m_{pck}	m_{loes}	m_{bdd}	t_{fd}	t_{loes}
Airport-P7	765	0.01	0.01	0.61	0	0.8
Airport-P8	27,458	0.57	0.26	9.82	1	43.92
Airport-P9	177,075	4.60	1.54	100.32	5	395.58
Blocks-7-0	38,688	0.13	0.09	0.61	0	2.89
Blocks-7-1	64,676	0.22	0.14	1.23	0	7.45
Blocks-7-2	59,167	0.20	0.13	1.23	0	5.56
Blocks-8-0	531,357	2.60	1.37	9.82	5	64.99
Blocks-8-1	638,231	3.12	1.51	4.91	6	87.77
Blocks-8-2	439,349	2.15	1.13	4.91	4	49.32
Blocks-9-0	8,000,866	43.87	19.13	39.29	90	1832.57
Blocks-9-1	6,085,190	33.37	16.54	39.29	73	1265.1
Blocks-9-2	6,085,190	33.37	15.10	39.29	75	1189.86
Blocks-10-0	103,557,485	629.60	271.02	130.84	MEM	110602
Blocks-10-1	101,807,541	618.96	275.43	283.43	MEM	112760
Blocks-10-2	103,557,485	629.60	283.75	130.84	MEM	110821
Depots-3	3,222,296	19.97	2.77	69.81	72	1174.48
Depots-4	135,790,678	1068.38	147.98	924.29	MEM	373273
Driverlog-6	911,306	3.58	0.81	0.61	21	144.74
Driverlog-4	1,156,299	3.72	0.83	0.61	20	195.22
Driverlog-5	6,460,043	23.10	4.45	1.23	162	1689.65
Driverlog-7	7,389,676	34.36	5.66	1.23	233	2735.61
Driverlog-8*	82,221,721	411.67	64.08	4.91	MEM	228181
Freecell-2	142,582	0.97	0.52	9.82	3	80.81
Freecell-3	1,041,645	9.19	4.47	39.29	25	904.13
Freecell-4	3,474,965	36.04	20.19	100.32	95	4321.72
Freecell-5	21,839,155	278.57	128.10	MEM	MEM	53941.8
Freecell-6*	79,493,417	1137.16	519.96	MEM	MEM	481452
Gripper-5	376,806	1.48	0.11	0.31	3	65.26
Gripper-6	1,982,434	8.74	2.06	0.31	20	466.55
Gripper-7	10,092,510	50.53	2.69	0.61	123	2894.97
Gripper-8	50,069,466	280.53	13.22	0.61	MEM	22720.9
Gripper-9	243,269,590	1479.00	133.92	1.23	MEM	410729
Microban-4	51,325	0.39	0.20	2.46	0.43	9.57
Microban-6	312,063	3.01	1.33	4.91	3.27	247.02
Microban-16	436,656	4.89	2.05	2.46	5	329.95
Microban-5	2,200,488	22.30	10.35	69.81	29.3	676.4
Microban-7	25,088,052	287.11	122.66	741.19	MEM	24574.1
Satellite-3	19,583	0.04	0.01	0.04	0	2.25
Satellite-4	347,124	0.95	0.12	0.08	13	74.5
Satellite-5	39,291,149	182.67	14.27	4.91	MEM	28580.9
Satellite-6	25,678,638	97.96	8.27	0.61	MEM	13684.6
Satellite-7*	115,386,375	591.47	37.96	2.46	MEM	380135
Travel-4	7,116	0.02	0.01	0.08	0.08	0.42
Travel-5	83,505	0.22	0.07	0.31	1.46	7.6
Travel-6	609,569	1.82	0.41	1.23	14.4	77.09
Travel-7	528,793	1.58	0.46	0.61	8.45	71.61
Travel-8	14,541,350	62.40	10.28	19.64	MEM	8178.87
Travel-9*	68,389,737	317.95	50.93	161.36	MEM	167795
Mystery-2	965,838	13.47	3.09	19.64	88.6	469.84
Mystery-4*	38,254,137	228.01	23.52	19.64	MEM	11674.1
Mystery-5*	54,964,076	563.49	150.90	130.84	MEM	290441
8-Puz-39944	181,438	0.78	0.42	4.91	1.08	26.49
8-Puz-72348	181,438	0.78	0.43	4.91	1.04	26.37
15-Puz-79*	23,102,481	176.26	102.32	558.08	MEM	17525.6
15-Puz-88*	42,928,799	327.52	188.92	771.70	MEM	71999.7

Table 1: Empirical results for a number of IPC domains, including (1) a comparison of peak-memory requirements between an idealized concatenated bit-string of packed states, LOES and BDD and (2) a comparison of runtimes between Fast Downward (FD) and LOES. $|O \cup C|$ is the number of states in Open and Closed before the goal layer, m_{pck}, m_{loes} and m_{bdd} are the respective peak memory requirements (in MB) of Packed, LOES and BDD storage components. t_{fd} and t_{loes} are the respective runtimes in seconds (or MEM if the process ran out of memory). Instances marked by * were those that exceeded our time allowance, in which case we included numbers from the largest layer both BDD and LOES processed. The naming of the 8-Puzzle instances is based on a lexicographic ordering of all such instances between 1 and 9!/2. The 15-Puzzle instances are from (Korf 1985).

than BDDs if the majority of the search space can be pruned away by a good heuristic function. Another key advantage over BDDs is that LOES allows one to easily associate arbitrary data to set elements without using pointers, which represent a significant storage overhead in symbolic as well as explicit-state search.

Instance	$loes$	$fd_{64\text{bit}}$	$fd_{32\text{bit}}$	$\frac{loes}{fd_{64\text{bit}}}$
Blocks-9-0	46.8	1460.3	990.9	0.03
Depots-3	17.8	737.2	553.9	0.02
Driverlog-7	37.9	3686.1	2142.9	0.01
Freecell-4	53.7	1092.9	866.6	0.05
Gripper-7	13.7	1720.5	1159.4	0.01
Microban-5	31.6	682.9	422.9	0.05
Satellite-4	18.6	101.8	64.2	0.18
Travel-6	22.1	225.1	124.0	0.10
Airport-9	11.4	171.9	152.8	0.07

Table 2: Peak allocated process memory for LOES and FD (64 and 32 bit binaries) in MB.

It can be observed from Table 1 that the runtime comparison is not in favor of LOES, which took about 10 and 20 times longer than FD on the larger instances both can solve. While certain overhead stems from employing LOES, a significant part is due to our current implementation, which can be substantially improved. To expand a node, our implementation performs a linear scan in the entire set of grounded operators to find the ones whose preconditions are satisfied; whereas FD uses a decision tree to quickly determine the set of applicable operators. Of course, the speed of our planner can be improved if the same decision tree were used. Another source of overhead is that our current implementation is not particularly optimized for bulk insertions, since we only used a small buffer to accommodate newly generated successors and whenever the buffer is full, it is combined with the next-layer LOES, which includes iterating over all previously generated states in the next layer.

Of course, FD does not store states in an ideally packed representation and instead uses highly time-efficient C++ STL-components for set storage. This results in significant space overhead. Table 2 gives a comparison of the peak process memory allocations for LOES and FD on the largest instance of each domain that both can solve. To show the influence of pointers, we also included numbers for a 32 bit binary of FD. As shown in the table, FD uses up to two orders of magnitude more RAM than LOES. Given FD's more advanced successor generator, it is somewhat surprising that LOES is only slower by a constant factor as the size of the instance increases in each domain.

Conclusion and Future Work

LOES shows good space efficiency for representing explicit state-sets of all sizes. It provides robust space savings even in traditionally hard combinatorial domains such as the n-puzzles. In particular, it defines a consecutive address-space over set elements, which allows space-efficient association of ancillary data to set-elements without addressing overhead. In theory, LOES should also offer good time efficiency, especially on larger problems, because its complexity of set-member testing depends only on the size of the state encoding and not on the size of the set. For the immediate future, an obvious application for LOES is pattern databases (PDBs). This class of heuristics traditionally depends on space-efficient representations with good look-up performance. LOES' static nature and potential construction overhead is of little concern for PDBs. We expect that LOES or similar representations have a future in duplicate detection and general set representations for heuristic search. We note that a sorted list of packed states can be transformed to LOES with little overhead and that LOES provides good compression even on small sets. Also a merge sort like construction where very imbalanced LOES merges are delayed until a similar-sized counterpart exists can help to achieve an amortized $O(\log n)$ construction. In both cases currently unused RAM can be used straightforwardly to speed up set construction, while the minimal peak-memory construction approach of our implementation can serve as a fallback as memory becomes scarce.

References

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35:677–691.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Edelkamp, S., and Helmert, M. 2001. The model checking integrated planning system (mips). *AI Magazine* 22(3):67–71.

Edelkamp, S., and Kissmann, P. 2008. Limits and possibilities of bdds in state space search. *KI 2008: Advances in Artificial Intelligence* 46–53.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

Jacobson, G. 1988. *Succinct static data structures*. Ph.D. Dissertation, Carnegie Mellon University Pittsburgh, PA, USA.

Jensen, R. M.; Bryant, R. E.; and Veloso, M. M. 2002. SetA*: An efficient BDD-based heuristic search algorithm. In *Proceedings of AAAI-2002*, 668–673.

Korf, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Lind-Nielsen, J.; Andersen, H.; Hulgaard, H.; Behrmann, G.; Kristoffersen, K.; and Larsen, K. 2001. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design* 18(1):5–23.

Munro, J., and Raman, V. 2001. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing* 31:762.

Zhou, R., and Hansen, E. 2004. Breadth-first heuristic search. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, 92–100.