Representing Pattern Databases with Succinct Data Structures

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

Abstract

In this paper we describe novel representations for precomputed heuristics based on *Level-Ordered Edge Sequence* (LOES) encodings. We introduce compressed LOES, an extension to LOES that enables more aggressive compression of the state-set representation. We evaluate the novel representations against the respective perfect-hash and binary decision diagram (BDD) representations of pattern databases in a variety of STRIPS domains.

Introduction

A key challenge for applying memory-based search heuristics such as pattern databases (Culberson and Schaeffer 1998) and merge & shrink abstractions (Drager, Finkbeiner, and Podelski 2006; Helmert, Haslum, and Hoffmann 2007) to domain-independent planning is succinct representation. The performance of these heuristics usually improves with the size of their underlying data, as well as the efficiency with which they can be accessed, with subsequent consequences to their memory requirements. The problem is exacerbated when employing best-first search algorithms with duplicate detection such as A*, since these algorithms are usually limited by the amount of available memory, a deciding factor for problem solvability.

Linear-space search algorithms such as IDA* (Korf 1985) use much less memory than A* but by forgoing duplicate detection they pay the price of extra node expansions to find optimal solutions. This time-space tradeoff pays off in domains with few duplicates such as the sliding-tile puzzles where IDA* easily outperforms A*, but many domains (e.g. multiple sequence alignment) are not conducive to this approach. Hence current state-of-the-art heuristic search planners such as Fast Downward (Helmert 2006), HSP^*_F and Gamer include full duplicate detection in their search algorithms.

Perfect hashing is a popular technique used to associate subproblems to precomputed solutions. Here an (*injective*) enumerating function assigns each subproblem a unique id that is typically used to address a record in a random access structure. However in domain-independent planning it is often infeasible to find a function that is also nearly *surjective*, particularly when dealing with complex subproblems. This quickly leads to excessive amounts of unused slots in the data array, relegating the approach to simple subproblems.

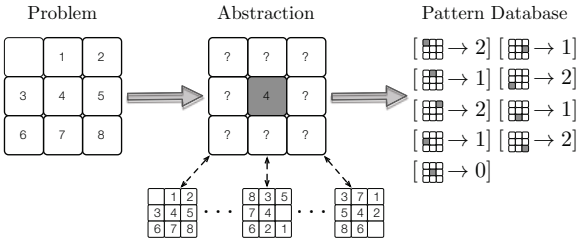

Figure 1: The 8-puzzle, its abstraction to tile 4 and the corresponding pattern database.

Another approach to representing pattern databases is to use BDDs (Edelkamp 2002). An (ordered and reduced) BDD is a canonical graph-based representation of a binary function, that through merging isomorphic sub-graphs is often very space efficient (Ball and Holte 2008). State-sets can be mapped to binary functions straightforwardly and BDDs have been successfully used in planning and model checking (Jensen, Bryant, and Veloso 2002). One problem for domain-independent planning is that the space efficiency of these approaches can vary widely depending on the structure of the underlying domain. Another problem is that BDDs are not well suited for associating data with individual states in a space efficient way. This is usually less of a problem for pattern databases, as often, a significant number of patterns is associated with the same value, but nevertheless this makes BDDs a challenge to use in more general memoization contexts.

In the following, we devise suitable pattern database representations for domain independent planning based on LOES encodings. We introduce a novel variant of LOES that trades off LOES support for efficient member ranking for better space efficiency in the context of inverse relation representations. Finally we give an initial evaluation of these representations in an independent planning setting.

Preliminaries

For brevity and clarity, we restrict ourselves to pattern databases for the remainder of this paper, but the ideas pre-

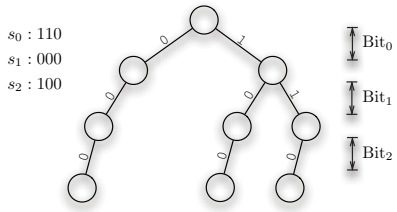

Figure 2: Three bit sequences and their induced prefix tree.

sented here should transfer rather straightforwardly to other memoization techniques. The idea behind a pattern database heuristic is to first create a (relatively) easy-to-solve abstraction of the original problem. A suitable abstraction must be interpretable as a (usually not invertible) function mapping original problem configurations to their abstract counterparts. An abstract problem configuration is referred to as a pattern. One then solves the abstract problem for all patterns and stores them with the associated costs of their optimal solutions in a database. Figure 1 gives an example. Informally, we abstract away the identities of all but tile 4. By associating each configuration of the 8-puzzle to the pattern with the matching position of tile 4, this can be interpreted as a many-to-one mapping function. Depicted on the right of Figure 1 then is the resulting pattern database.

We will now lay out central concepts of the LOES representation. Our overview is likewise limited to the facilities necessary for their representation (c.f. (Schmidt and Zhou 2011) for an in-depth discussion of the techniques behind the encoding). We assume that any pattern (or abstract state) in the database can be encoded in m bits for a given heuristic. Such sets of patterns can be *bijectively* mapped to edge-labeled binary trees of depth m with labels $false$ and $true$ by mapping each pattern to a path from root to leaf with an edge at tree-depth d corresponding to the value of the bit at offset d in the pattern's bit-string. In this way every unique pattern results in a unique path and can be reconstructed by the sequence of edge-labels from root to leaf. Henceforth, we refer to these trees as prefix trees. An example is given in figure 2.

A worthwhile preprocessing step is to determine a permutation on the encoding of the patterns that minimizes this prefix tree (see Figure 3). Methodically sampling the (abstract) state-space and a greedy entropy search through the permutation space based on these samples usually gives good results with little computational effort.

LOES encoding

LOES allows us to represent prefix trees in (strictly) less than two bits per edge. 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 4 shows how this allows us to encode an example set in a single byte. The encoding results in a bit-string of between $2n$ and $\approx 2nm$ bits for a set of n states with m bit representation, with the average case usually close to the lower bound (cf (Schmidt and Zhou 2011)).

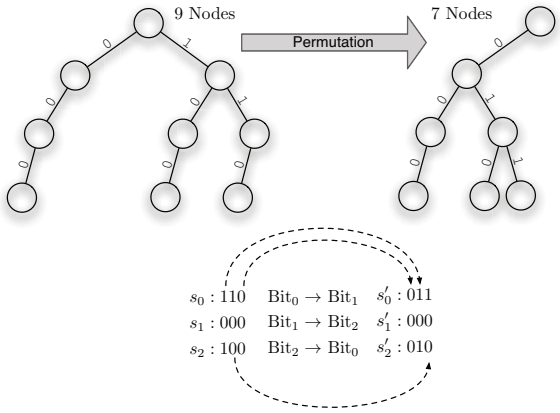

Figure 3: Permuting the bit-order of the encoding can lead to smaller prefix-trees.

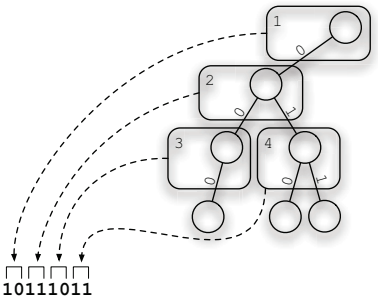

Figure 4: Level-ordered edge sequence for the example set.

LOES navigation

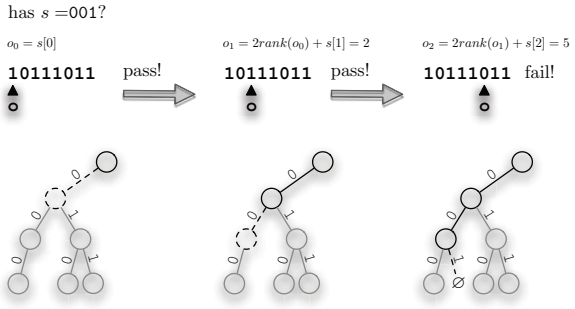

Figure 5: path offset computation for `001` in the example set. At each level, we compute the offset of the corresponding edge-presence bit, test the bit, continue on pass and return $\perp$ on a fail.

In the LOES encoding, the presence bits for the $false$ and $true$ edges of the node some edge at offset o points to can be found at offsets $2rank(o)$ and $2rank(o)+1$, where $rank(i)$ is a function that gives the number of set bits in the sequence

up to (and including) offset i. This follows from the level-order encoding - each preceding edge (with the exception of the leaf level) results in a preceding edge-pair record for the respective target node of that edge. 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). This property allows efficient navigation over the encoded structure.

Rank. Using a two-level index, which logically divides the LOES into blocks of 2^{16} bits and sub-blocks of 512 bit, $rank$ can be computed in constant time. 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 total index size is around 3.3% the size of the LOES code (see Equation 1). Using simple address translations this reduces the computation to the sum of two table lookups and a straightforward population count in the respective 512 bit sub-block.

$$\underbrace{\frac{64}{2^{16}}}_{\text{block index}} + \underbrace{\frac{16}{2^{9}}}_{\text{sub-block index}} \approx 0.0323 \qquad (1)$$

Member test. The path-offset function (Algorithm 1) navigates through the LOES from the root according to the label-sequence 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 5. A set contains a state, if and only if its path interpretation corresponds to a valid path through the prefix tree and path-offset returns $\neq \perp$.

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 ⊥;
```

Algorithm 1: path-offset

Member index. Other than efficiently computing member-set tests, the encoding allows to associate consecutive ids $\in \{\perp, 0, \ldots, n-1\}$ for each state in the set. The idea is to compute the rank of the path-offset of a state and normalize this to the $[0, n)$ interval by subtracting the rank of the last offset of the last but one layer +1. Algorithm 2 gives the pseudo-code and figure 6 shows this for our example set.

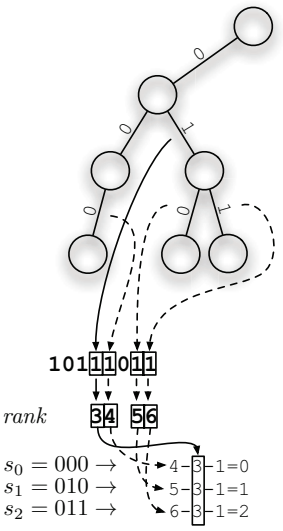

Figure 6: Index mappings for all states in the example set. We subtract the rank+1 (of the offset) of the last edge in the last-but-one level from the rank of the path-offset of an element to compute its index.

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

```
o ← path-offset(state);
if o = ⊥ then
    return ⊥;
a ← rank_LOES(o);
b ← rank_LOES(levelOffsets[m − 1] − 1);
return a − b − 1;
```

Algorithm 2: member index

LOES Construction

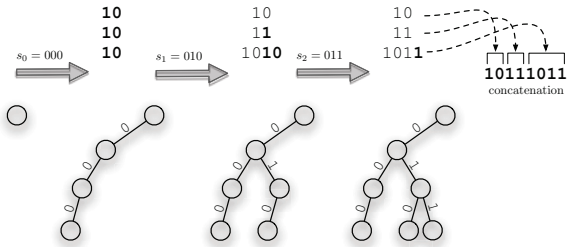

Figure 7: Construction states when building a LOES code from the example states with algorithm 3.

We end this digest of LOES by discussing how to construct the code from a sequence of lexicographically ordered states. We start with an empty bit-sequence for each layer of

```
Input: s a bitsequence of length m
Data: treelevels an array of bitsequences
Data: s' a bitsequence of length m or ⊥
if s' = ⊥ then
    depth ← −1;
if s' = s then
    return;
else
    depth ← i : ∀j < i, s[j] = s'[j] ∧ s[i] ≠ s'[i];
    treelevels[depth]_lastBit ← true;
for i ← depth + 1 to m − 1 do
    if s [i] then
        treelevels[i] ← treelevels[i] ∘ ⟨01⟩;
    else
        treelevels[i] ← treelevels[i] ∘ ⟨10⟩;
s' ← s;
```

Algorithm 3: add state

the tree. Algorithm 3 shows how these sequences are manipulated when adding a new state. For the first insertion, we merely append the corresponding records on all levels. For the following insertions we compute the length of the shared prefix between the inserted state s' and the last state in the code s and set d to the offset of the first differing bit. We set the last bit of sequence for level d to $true$ and then append records according to s to all lower levels. Duplicates (i.e. $s = s'$) are simply ignored. Once we are done with inserting states, the LOES code is constructed by forming the concatenation of the sequences in ascending order of their levels.

PDB Representations

Having introduced the basics of LOES, we now turn our attention to the representation of pattern databases. For brevity we will not concern ourselves in detail with pattern selection, domain abstraction and the regression search, but will assume a pattern database has already been computed and exists as some collection of pattern-value pairs. For in-depth coverage of these interesting topics, we would like to point the reader to (Haslum et al. 2007; Helmert, Haslum, and Hoffmann 2007).

Combined Layer Sets

Inverse Relation

A basic representation is to convert all patterns into a LOES code. LOES associates a unique id with every unique pattern in the range $\{0, \ldots, |\text{PDB}| - 1\}$ which we use as an offset to store the associated values in a packed bit-string where each record comprises of the minimal amount of bits necessary to discern between the occurring (in the PDB) values. Computation of the heuristic then comprises of determining the id of the pattern using algorithm 2, and get the value by interpreting id as an offset into the packed bit-string.

Especially in unit-cost search, the number of patterns in a PDB usually by far outstrips the number of different values.

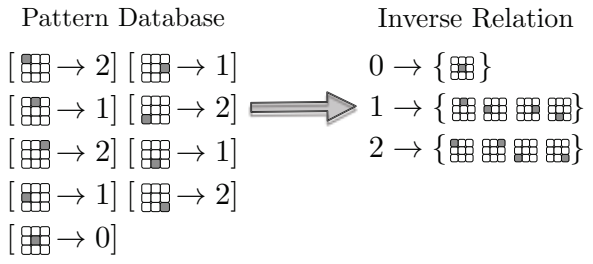

Figure 8: PDB for tile 3 of the 8-puzzle and its inverse relation.

We can avoid associating this repetitive data with individual patterns by storing the inverse of the heuristic function. In general, heuristics are not *injective*, hence a well-defined inverse does not exist. Instead, the *inverse relation* (a left-total relation, where every input is associated with multiple outputs) is stored (see figure 8 for an example). The heuristic function is then computed through consecutive tests of the pattern against each of the pattern-sets and upon a hit, returning that set's associated value. Note, that due to the function property of the heuristic, these sets are pairwise disjunct. If furthermore, the heuristic is a total function (i.e. the union over all pattern sets comprises the entire abstract pattern space), we can remove the largest of the sets and denote its associated value as a default which we return if the test against all remaining sets fail. The idea is to represent each set as a LOES code. A further optimization is to keep track of the success probabilities of the member-tests over time and query the sets in descending order of these probabilities.

Compressed LOES

```
Input: state a bitsequence of length m
Output: offset an offset into cLOES
Data: cLOES an encoded state-set
offset ← 0;
for depth ← 0 to m − 1 do
    if cLOES[offset, offset + 1] = 00 then
        return offset;
    if state[depth] then
        offset ← offset + 1;
    if cLOES[offset] then
        if depth = m − 1 then
            return offset;
        else
            offset ← 2rank_cLOES(offset);
    else
        return ⊥;
```

Algorithm 4: comp-path-offset

For the inverse relation representation, we do not need to associate any information with individual states, but only be able to compute set membership. If we encounter a root of a complete subtree during a descend through the prefix tree,

we already know that the element in question is present. To exploit this, we developed a variant of LOES, called compressed Level Order Edge Sequence (cLOES), that allows us to omit complete subtrees from the structure. The idea is straightforward - we use the remaining code-point namely 00 (i.e. no further edge at this node) to denote a root of a complete subtree. Note that this does *not* violate the edge-index child-record-position invariant of LOES. As algorithm 4 shows, the changes to member tests are minimal - whenever we reach a new record, we first test if it denotes a complete subtree (i.e. equals 00) and if so return the current offset. Else the algorithm behaves analogously to LOES.

```
Input: s a bitsequence of length m
Data: codes an array of bitsequences
Data: s' a bitsequence of length m or ⊥
if s' = ⊥ then
    depth ← −1;
if s' = s then
    return;
else
    depth ← i : ∀j < i, s[j] = s'[j] ∧ s[i] ≠ s'[i];
 8  if depth = m − 1 then
        codes[depth]]_lastRec ← 00;
        for i ← depth − 1 to 0 do
            if codes[i]_lastRec = 11
            ∧codes[i + 1]_last2Recs = 0000 then
                codes[i]_lastRec ← 00;
                codes[i + 1].popRecord();
                codes[i + 1].popRecord();
            else
                break;
18
    else
        codes[depth].lastBit ← true;
        for i ← depth + 1 to m − 1 do
            if s [i] then
                codes[i] ← codes[i] ∘ ⟨01⟩;
            else
                codes[i] ← codes[i] ∘ ⟨10⟩;
    s' ← s;
```

Algorithm 5: comp-add-state

Tree construction also changes slightly from LOES. Algorithm 5 gives the procedure along with cLOES modifications (lines 7 to 17). If an insertion generates an 11 record on the leaf level, we convert this to a 00 record. We then run a simple bottom-up pattern-matching algorithm over (the ends of) all level codes. The pattern to match is where the bit-string on some level i ends in 11 and the bit-string on the lower level in 0000. On a hit we prune the last four bits of the lower level and change the record of the higher level into 11. The intuition behind this is simple - whenever a record sports two edges pointing to complete subtrees, the record is a root-node of a complete subtree.

Empirical Evaluation

Our evaluation setup consisted of a preprocessor for converting PDDL input files into multivalued problem descriptions similar to Fast Downward's preprocessor. The difference is that our preprocessor outputs additional at-most-one constraints covering the problem variables. They come in the form of lists of variable-assignment tuples and are interpreted such that for any valid state, at most one of the tuples in every list holds true. For the original problem, these constraints add no additional information over what is encoded in the multi-valued description - that is, no successor of the initial state generated through the operator set will violate these constraint. The problem is that these implicit constraints are lost due to abstraction by projecting to a subset of variables.

Consider the multi-valued encoding generated by Fast Downward's (and our) preprocessor for the N-puzzles. It comprises of one variable for each tile denoting its position. There are operators for every viable pairing of the blank tile and neighboring non-blank. Each such operator has the specific positions of the blank and the tile as precondition with their switched positions as effect. As tiles start out on distinct positions in the initial state, the constraint that no two tiles can occupy the same position is implicitly upheld through the operator set. Once even a single variable is projected away (which results in the removal of its references from all operator preconditions and effects) that constraint is violated, creating a non *surjective* abstraction (i.e. there are viable patterns in the abstraction, that have no counterpart in the original problem).

This creates two problems. The lesser one is an often exponential increase in the size of the pattern database. The greater one is the severe reduction in quality of the resulting heuristic. If one, say, projects on 7 variables from the 15-puzzle, the resulting database will comprise ≈ 270 million patterns, but as tiles can move "through" each other will carry no more information than the manhattan distance of these 7 tiles. Note, that this does not affect the *admissibility* of the heuristic. Evaluating these "redundant" constraints in the abstract space allows us to mitigate this problem by upholding additional constraints (see also (Haslum, Bonet, and Geffner 2005)).

The translation process is followed by a rule based system selecting variables for one or more PDBs. Both of these components are experimental at this point which somewhat limited the scope of our evaluation. Then the PDBs would be constructed through a regression search and encoded in one of five representation forms.

Perfect Hashing (PH) The perfect hash function maps each possible assignment vector (of the abstract problem) to a unique id given by its lexicographic rank. Ids are used for addressing packed records holding the associated values.

Binary Decision Diagram (BDD) The PDB is stored as an inverse relation with each set represented as a BDD as described above. Common subgraphs are shared between sets. We used the buddy package, a high performance implementation from the model checking community for

our evaluation.

LOES Analogous to PH. The perfect hash function is implemented through a LOES set of all *occurring* patterns and its member-index function.

Inverse Relation LOES (IR LOES) Analogous to BDD. Each set is represented as a LOES. All sets use the same encoding permutation.

Inverse Relation compressed LOES (IR cLOES) Analogous to BDD. Each set is represented as a cLOES with a specific encoding permutation.

The PDBs were then used in an A* search. The "Pipesworld Tankage", "Driverlog" and "Gripper" instances were run on a 2.2GHz Intel Core processor running Mac OS 10.6.7 with 8 GB of memory. For the 15-Puzzle STRIPS instances, we used a 3.3GHz Xeon processor with 4GB of memory.

Table 1: Total PDB size, solution length and complete search time (parsing, analysis, PDB construction and search) for the "Pipesworld Tankage" (pt), "Driverlog" (dl) and "Gripper" (gr) instances.

#	size	sl	t_{PH}	$t_{IR\ LOES}$	$t_{IR\ cLOES}$	t_{LOES}	t_{BDD}
pt1	67144	5	1.4	1.4	1.7	2.6	3.3
pt2	3559	12	0.2	0.1	0.2	0.2	0.2
pt3	38204	8	1.8	1.8	2.0	2.1	2.7
pt4	85422	11	5.0	4.9	5.7	5.9	7.4
pt5	212177	8	9.9	10.2	10.8	11.6	16.7
pt6	113364	10	9.0	9.3	9.8	9.8	11.6
pt7	13620	8	4.2	4.3	4.1	4.2	4.7
pt8	35307	11	11.5	14.4	11.9	12.8	16.6
dl1	30375	7	0.3	0.4	0.6	0.6	0.8
dl2	339750	19	8.9	9.2	11.2	12.6	16.3
dl3	1766250	12	37.4	35.7	38.5	51.3	75.2
dl4	2466625	16	125.5	118.0	120.7	151.9	179.4
dl5	1800000	18	54.9	55.5	57.1	65.6	83.5
dl6	4478976	11	266.5	267.7	267.3	294.2	352.0
dl7	3779136	13	333.1	337.7	334.7	359.4	417.9
gr1	600	11	0.0	0.0	0.0	0.0	0.0
gr2	4604	17	0.0	0.1	0.1	0.1	0.2
gr3	30320	23	0.4	0.6	1.2	1.0	1.4
gr4	181428	29	3.6	3.9	8.1	7.5	9.9
gr5	1016072	35	28.5	25.9	37.3	52.3	63.6
gr6	1460128	41	51.7	63.7	78.2	90.4	190.5
gr7	1975008	47	136.6	237.8	277.8	206.3	746.3
gr8	2582788	53	574.9	1187.4	1346.0	757.7	3751.0

Pipesworld Tankage

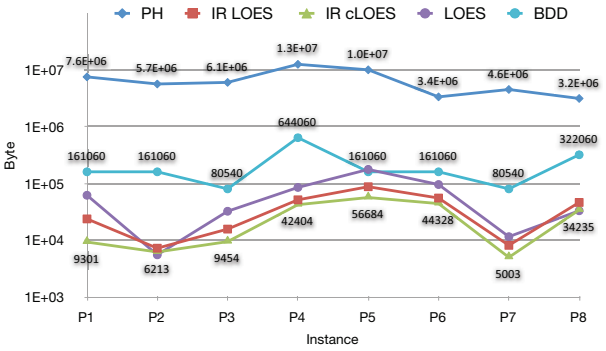

Figure 9: Size of the PDB representations in bytes for the "Pipesworld Tankage" instances.

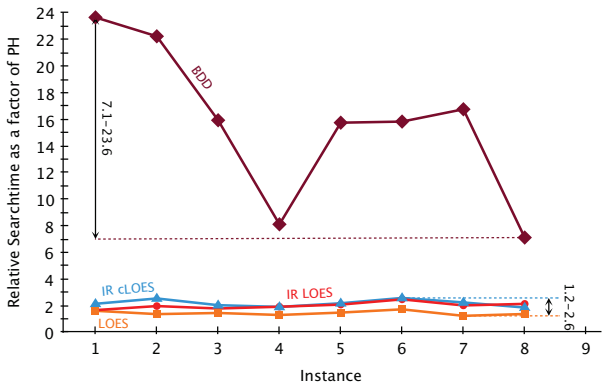

Figure 10: Relative search time as a multiple of PH for the "Pipe Tankage" instances.

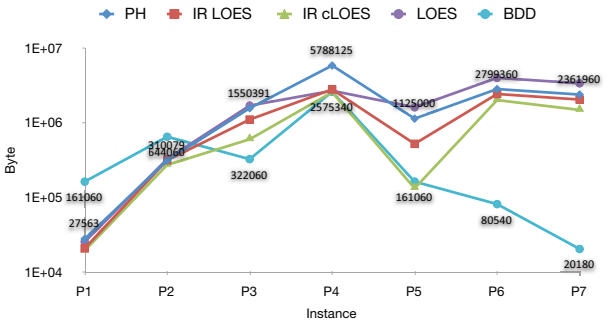

Figure 11: Size of the PDB representations in bytes for the "Driverlog" instances.

The IPC4 "Pipesworld Tankage" domain models the problem of transporting oil derivatives through pipeline segments connecting areas that have limited storage capacity due to tankage restrictions for each product. The additional constraints made explicit by the preprocessor state that for any pipe, there can only be one batch that is the nearest to a source area and one batch that is the nearest to a destination area. The analysis component generated single PDBs for all instances. The PDBs are relatively small and retain a good amount of the original problem's constraints. This shows in the sizes for the different representations (see Figure 9) where BDD outperforms PH by between one to two orders of magnitude, with the LOES versions besting this by another order of magnitude.

On the time dimension (see Figure 10), LOES only performs marginally worse than PH while the IR variants take about twice as long. BDD performance varies considerably and performs a good order of magnitude worse than PH and the LOES encodings.

Driverlog

"Driverlog" is an example where our preprocessing fails to uncover any explicit constraints over those encoded in the multi-valued variables. This results in PDBs comprising of

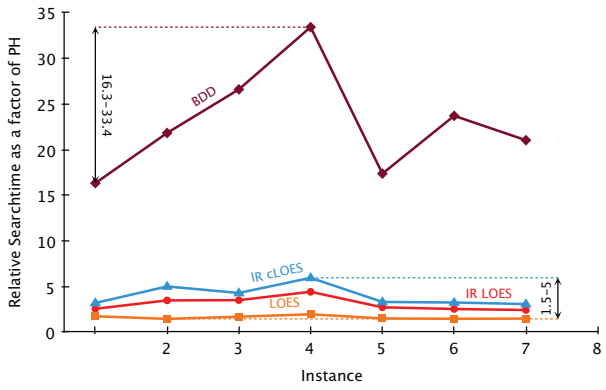

Figure 12: Relative search time as a multiple of PH for the "Driverlog" instances.

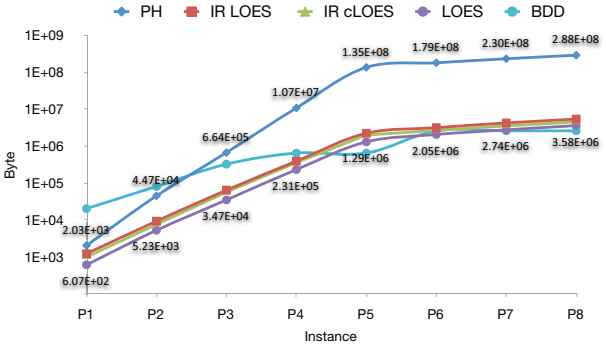

Figure 13: Size of the PDB representations in bytes for the "Gripper" instances.

all possible abstract patterns with very low quality. It is also a domain that is quite amendable to BDD representation. This shows in the space comparison (see Figure 11), where the BDD shines on the large, multi-million pattern instances (that are scarcely subdivided by the IR representation), actually taking up an order of magnitude less space than on the smaller instances. Remarkably the IR LOES variants still manage to outperform PH by a factor of two to three. LOES predictably performs worse as its packed store is nearly the same size as PHs (the difference stems from the fact, that it can omit storing the largest equal-value subset of patterns in PBD and denote the corresponding value as its default). The runtime comparison (see Figure 12) paints a similar picture, as the representations' look-up cost is similar to the"Pipesworld Tankage" instances.

Gripper

The "Gripper" domain models a mobile robot that can use its two grippers to pick up and put down balls, in order to move them from one room to another. In this domain, the preprocessor picked up the implicit constraint that no object can be in both grippers at the same time. The variable selection logic constructed PDBs comprising of the fluents for

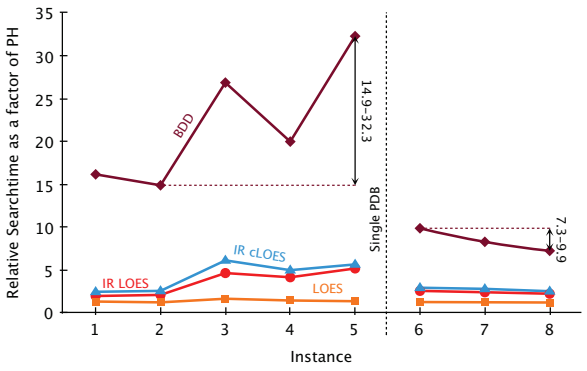

Figure 14: Relative search time as a multiple of PH for the "Gripper" instances.

the gripper states, the location of the robot and goal qualified balls. A rule was in place that would split PDBs as the abstract state space grew too large. As the multiple resulting PDBs were not additive they were combined by taking the maximum of their heuristic values. This happened beginning with instance 6, mitigating the growth of the PDBs (see Figure 13).

"Gripper" is one of the domains where BDDs are known to perform extremely well. Still it outperformed IR cLOES in storage efficiency only in instances 5 and 8, when the PDBs where about 1 and 2.6 million patterns in size. PH consistently required around 2 orders of magnitude more storage on the larger instances. The runtime comparison (see Figure 14) paints an interesting picture. For the smaller PDBs, PH is about 1.3 (LOES) to 5 (IR cLOES) times faster than the LOES versions. As the pattern databases grow the advantages from the quick addressing shrink considerably, probably due to increased cache misses. Again the more complex matching in BDDs is a good order of magnitude slower.

15-Puzzle

The 15-Puzzle is a classic combinatorial search benchmark. It is also a token problem for PDB heuristics. Our preprocessor here manages to extract constraints ensuring that no two tiles can occupy the same position. Also the analysis component manages to extract multiple, *additive* PDBs by excluding the blank and selecting tiles up to its pattern space size limit (up to 6 variables in this domain, hence an additive 6-6-3 PDB). Note that these PDBs are still noticeably weaker than the handcrafted and blank-compressed additive PDBs typically employed in domain-specific sliding-tile puzzle solvers. We run our planner over Korf's 100 random instances (Korf 1985), which are challenging for domain-independent planners (e.g., the state-of-the-art Fast Downward planner using merge & shrink heuristic with an abstraction size of 10K nodes cannot solve the easiest instance within 96 GB of RAM). It is also a permutation problem, which is known to be problematic for the type of redundancy-elimination techniques employed by LOES and

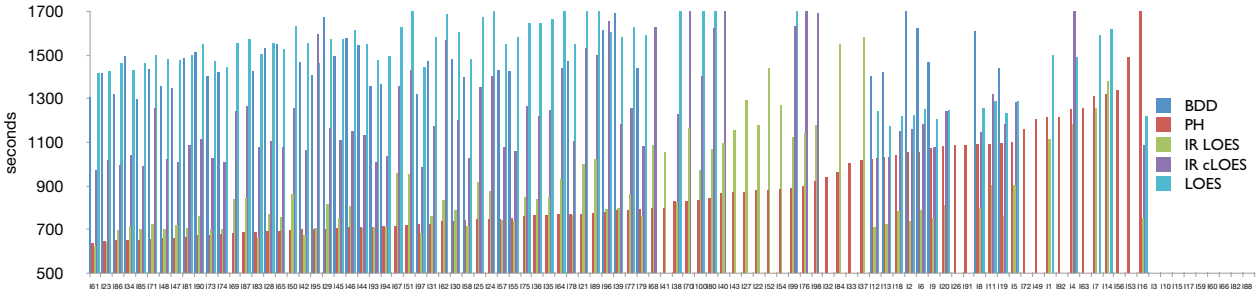

Figure 15: Planning time (parsing, analysis, PDB creation and search) over Korf's 100 instances of the 15-puzzle. *I*23 stands for instance 23. If a representation has no bar for an instance, it failed to terminate successfully in 30 minutes.

BDDs. We ran all instances with a hard 30 minute cut-off timer. Table 2 gives the results. Here PH fared the best, thanks to its very quick PDB lookups. While the LOES variants offered a noticeable relative reduction in PDB size, the absolute differences were relatively small. The results would probably change if the analysis component allowed larger PDBs (to the detriment of the BDD based representation). Figure 15 gives an overview of the total planning time over all hundred instances.

Table 2: Number of instances solved, PDB size and relative search speed over Korf's hundred 15-puzzle instances.

Rep.	$\#_{solved}$	size (MB)	$\frac{t_x}{t_{PH}}$
PH	91	20.0	1.00
BDD	40	117.6	0.11
IR LOES	82	11.4	0.36
IR cLOES	70	9.6	0.23
LOES	68	11.2	0.49

Conclusion and Future Work

We believe techniques such as LOES offer exciting opportunities to better exploit dynamic programming and other memoization techniques in domain-independent planning. The approach allows for quite efficient representation of strong, precomputed heuristics. The very basic domain analysis we employed in our evaluation can only give a hint of the potential for ad-hoc abstraction in heuristic search. Of interest is also LOES' impedance match with BDDs. The inverse relation representation straightforwardly allows to adaptively interchange BDD and LOES based representations of state-sets. In this way, a domain-independent planner can leverage the superior efficiency of BDDs in appropriate domains while mitigating their lack of robustness by falling back to a LOES-based representation.

References

Ball, M., and Holte, R. 2008. The compression power of symbolic pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2–11.

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

Drager, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed model checking with distance-preserving abstractions. In *Model checking software: 13th International SPIN Workshop, Vienna, Austria, March 30-April 1, 2006: proceedings*, 19. Springer-Verlag New York Inc.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Proceedings of the 6th International Conference on AI Planning and Scheduling (AIPS-02)*, 274–283.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, 1007. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, 1163. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS*, volume 2007, 176–183.

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

Jensen, R.; Bryant, R.; and Veloso, M. 2002. SetA*: An efficient bdd-based heuristic search algorithm. In *Proceeding of the 18th National Conference on Artificial Intelligence (AAAI-02)*, 668–673.

Korf, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*.

Schmidt, T., and Zhou, R. 2011. Succinct set-encoding for state-space search. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*.