On Transposition Tables for Single-Agent Search and Planning: Summary of Results

Yuima Akagi
Tokyo Institute of Technology

Akihiro Kishimoto
Tokyo Institute of Technology
and JST PRESTO

Alex Fukunaga
University of Tokyo

Abstract

Transposition tables are a well-known method for pruning duplicates in heuristic search. This paper presents a detailed analysis of transposition tables for IDA*. We show that some straightforward implementations of IDA* with transposition tables (IDA*+TT) can result in suboptimal solutions being returned. Furthermore, straightforward implementations of IDA*+TT are not complete. We identify several variants of IDA*+TT which are guaranteed to return the optimal solution, as well as a complete variant. An empirical study shows that IDA*+TT can significantly improve upon the performance of A* in domain-independent planning.

1. Introduction

Best-first search strategies such as A* (Hart, Nilsson, and Raphael 1968) are widely used for solving difficult graph search problems, but a significant limitation of A* is the need to keep all of the generated nodes in memory. An alternative approach for exploring the search space in a best-first manner without storing all nodes includes linear-space algorithms such as IDA* (Korf 1985). IDA* performs a series of depth-first searches with a cutoff bound, such that on each iteration, states with a cost less than or equal to the bound are expanded. A major issue with IDA* when searching a graph is the re-expansion of duplicate states reached via different paths in the graph, which can result in a tremendous amount of redundant search.

One method for detecting and pruning duplicate nodes in IDA* is a transposition table, which caches information about previously generated nodes. When a node is generated, this cache is consulted to detect and prune duplicate nodes. As far as we know, transposition tables (TT) for single-agent search were first proposed in (Reinefeld and Marsland 1994) as one of the components of an "enhanced IDA*". Although the use of transposition tables in IDA* has been reported in a number of domains since then, there has not been an in-depth analysis of IDA* using a TT **(IDA*+TT)** in the literature. Although the basic idea of a transposition table is simple, it turns out that there are subtle but very important algorithmic details which affect the admissibility and completeness of a search algorithm that uses a TT. In addition, the choice of replacement policy for the TT has a significant impact on the performance of IDA*+TT.

```
algorithm Iterative Deepening;
1: $bound := h(root); path := [root]; solved := 0$;answer:=[];
2: repeat
3:     $bound :=$ DFS*$(root, bound, path)$;
4: until $solved \vee bound = \infty$;
5: if $solved$ then
6:     return $path$;
7: else
8:     solution doesn't exist!;
9: end if
```

Figure 1: Iterative Deepening template. **DFS*** calls one of the recursive search functions described in the text.

In this paper, we investigate transposition tables for IDA*. We first show that some straightforward implementations of transposition tables can result in IDA* returning suboptimal solutions, as well as failing to terminate correctly when there is no solution (incompleteness). These problems are particularly prone to occur when arbitrary replacement strategies are used to replace or retain elements in the TT when the table is full. We identify IDA*+TT algorithms which are guaranteed to find the optimal solution (regardless of replacement policy), but are incomplete. We then propose an IDA*+TT algorithm which is complete. Then, we discuss and propose replacement policies for transposition tables, and empirically demonstrate the effectiveness of IDA*+TT in domain-independent planning. We implemented IDA*+TT in a recent version of the state-of-the-art, Fast-Downward (FD) sequential optimal planner (Helmert, Haslum, and Hoffmann 2007), and show that IDA*+TT can significantly outperform the standard A* algorithm in FD.

2. IDA* With Transposition Tables

IDA* performs an iterative-deepening search, as shown in Figures 1 and 2, where each iteration performs a depth-first search until the cost (f-value) exceeds *bound* (Fig 2). Given an admissible heuristic function, IDA* returns a minimal-cost solution, if a solution exists (Korf 1985).

When the search space is a graph, IDA* will regenerate duplicate nodes when there are multiple paths to the same node. A transposition table (TT) is a cache where the keys

```
function DFS($n, bound, path$): real
 1: if $n$ is a goal state then
 2:     $solved := true$; $answer := path$; return $(0)$;
 3: end if
 4: if successors($n$) $= \emptyset$ then
 5:     $new\_bound := \infty$ ;
 6: else
 7:     $new\_bound := \min\{\text{BD}(m) | m \in \text{successors}(n)\}$;
 8: end if
 9: return $(new\_bound)$;
```

where BD(m) :=

Case 1: ∞, if $path + m$ forms a cycle

Case 2: $c(n, m) + \text{DFS}(m, bound - c(n, m), path + m)$, if $c(n, m) + h(m) \leq bound$

Case 3: $c(n, m) + h(m)$, if $c(n, m) + h(m) > bound$

Figure 2: DFS for standard IDA*

are states and the entries contain the estimated cost to a solution state. The TT is usually implemented as a hash table. During the search, we compute the hash value for the current state (if we use an *incrementally* computable hash function such as (Zobrist 1970), this is at most several XOR operations per node), and then perform a hash table lookup to check if the current state is in the TT. While this imposes an additional overhead per node compared to standard IDA* in order to prune duplicates, this tradeoff can be favorable in applications such as domain-independent planning where duplicate nodes are common, and this overhead is small compared to state generation and heuristic computation.

Since a TT has finite capacity, a *replacement policy* can be used to manage this limited capacity, i.e., determine how/which entries are retained or replaced when the table is full. In the analysis below, the replacement policy is assumed to behave arbitrarily, so the results are independent of replacement policy.

We now analyze some properties of IDA*+TT. We assume that the search space is a finite, directed graph, which may contain cycles.

Definition 1 *A search algorithm is* **admissible** *if it returns the minimum cost solution in finite time (assuming one exists).*

A key property of search algorithms is whether it is guaranteed to be able to find an optimal solution. Whether a particular instance of IDA*+TT is admissible or not depends on a subtle combination of algorithmic details (specifically, the interaction among the backup policy, cycle detection mechanism, and TT replacement policy), as well as problem characteristics (i.e., whether the heuristic is consistent).[1]

[1]For example, the 15-puzzle implementation in (Reinefeld and Marsland 1994) combines IDA*+TT with a consistent heuristic based on Manhattan distance, a TT replacement strategy based on search depth, and a move generator eliminating a move placing a piece back to the blank where that piece was located in the previous state (such a move immediately creates a cycle).

```
function DFSTT1($n, bound, path$): real
 1: if $n$ is a goal state then
 2:     $solved := true$; $answer := path$; return $(0)$;
 3: end if
 4: if successors($n$) $= \emptyset$ then
 5:     $new\_bound := \infty$;
 6: else
 7:     $new\_bound := \min\{\text{BD}(m) | m \in \text{successors}(n)\}$;
 8: end if
 9: store $(n, new\_bound)$ in $TT$;
10: return $(new\_bound)$;
```

where BD(m) :=

Case 1: ∞, if $path + m$ forms a cycle

Case 2: $c(n, m) + \text{DFSTT1}(m, bound - c(n, m), path + m)$, if $c(n, m) + \text{Lookup}(m) \leq bound$

Case 3: $c(n, m) + \text{Lookup}(m)$, if $c(n, m) + \text{Lookup}(m) > bound$

```
function Lookup($m, TT$): real
 1: if $m$ is is in $TT$ then
 2:     return $esti(m)$;
 3: else
 4:     store $(m, h(m))$ in $TT$
 5:     return $h(m)$
 6: end if
```

Figure 3: DFSTT1 - straightforward (but inadmissible) extension of DFS using a transposition table; uses auxiliary Lookup function

Let us consider DFSTT1 (Fig 3), a straightforward extension of DFS which uses a a transposition table. The major difference is that in the computation of $BD(m)$ (the lower bound on the cost of reaching a solution from m), calls to the heuristic function h are replaced with calls to $Lookup$, which either returns the stored estimate for a node (if the entry exists), or computes, stores, and returns the estimate. Since cycle detection is important in applications such as domain-independent planning, DFSTT1 incorporates a general, cycle detection mechanism which detects cycles of arbitrary length.

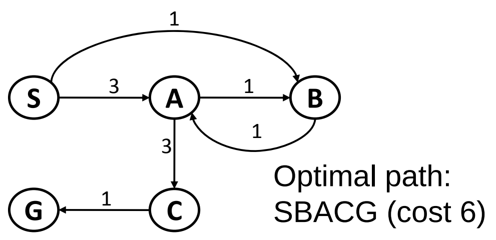
If the capacity of the TT is unlimited (infinite memory), then, with a consistent heuristic it is straightforward to show that DFSTT1 is guaranteed to return the optimal solution (if one exists).

Proposition 1 *Given a consistent heuristic, IDA* using DFSTT1 with an infinite capacity TT is admissible.*

On the other hand, if the TT capacity is finite, it turns out that DFSTT1 can return a suboptimal solution, depending on the replacement policy.

Proposition 2 *Given a consistent heuristic, IDA* using DFSTT1 with a finite-capacity TT is not admissible (for some replacement policies).*

Proof: Consider the search graph in Fig 4. Assume that the TT replacement policy is such that the estimates for A and C are never stored (this could be part of a larger graph which exhausts TT capacity), so that their h-value is always computed and used, and that B is initially not stored. The cutoff



$c(S,A) = c(A,C) = 3$

$c(S,B) = c(A,B) = c(B,A) = c(C,G) = 1$

$h(S) = 2, h(A) = 2, h(B) = h(C) = 1$

Successor ordering for S: A, B
Successor ordering for A: B, C

Figure 4: Counterexample for Proposition 2

bound starts at ($bound = h(S) = 2$), and increases to 4, then 5. Consider the iteration with $bound = 5$. The cycle $[S, A, B, A]$ is detected. At this point, (B, ∞) is added to the TT – assume that the replacement policy never replaces this entry. Next, when the path $[S, B]$ is generated, we compute $f(B) = 1+esti(B) = \infty$ and back up. On the next iteration ($bound = 7$), the solution path $[S, A, C, G]$ is found, and the algorithm terminates and returns this suboptimal path. □

Note that in this counterexample, the interaction between the TT cycle detection mechanism and replacement strategy is responsible for the incorrect behavior.

The definition of DFSTT1 in Fig 3 does not specify a TT replacement policy. While there exist replacement policies such that IDA*+DFSTT1 always returns the optimal solution, this is not quite satisfactory, because the TT is essentially a cache, and it is preferable to identify IDA*+TT algorithms which are admissible regardless of TT replacement policy.

In fact, even with unlimited memory and no replacement, DFSTT1 is not admissible if the heuristic is inconsistent.

Proposition 3 *Given an admissible, inconsistent heuristic, IDA* using DFSTT1 is not admissible.*

This can be seen by replacing $h(S) = 2$ with $h(S) = 5$ in Fig 4 and simulating an execution similar to that for Proposition 2.

3. Admissible IDA* with a Transposition Table

The main problem with DFSTT1 is the interaction between the transposition table and cycle detection mechanism, which can result in an incorrect value (∞) being stored in the TT. This is further complicated by the use of a replacement strategy when TT capacity is limited. We can correct this problem by keeping track of the lower bound to return to the parent call ($bound$) and the estimate to be stored in the TT ($esti$) separately. This modified algorithm, DFSTT2, is shown in Fig 5. Note that DFSTT2 returns a pair of values, (new_esti, new_bound) (Lines 11,2). The ET computation uses the first return value (index "[0]"), and the BD computation uses the second value (index "[1]").

Theorem 1 *Given an admissible heuristic function, IDA*+DFSTT2 is admissible.*

Proof Sketch: Assume (without loss of generality) a search graph with non-negative edge costs and a non-negative heuristic function. Assuming a solution exists, let C^* be the cost of an optimal solution. The following properties hold:
(P1) DFSTT2($root, bound, [root]$) terminates.
(P2) For any node n, $0 \leq g_{min}(n) + \text{esti}(n) \leq COST^*(n)$ where $g_{min}(n)$ is the smallest g-value among multiple paths to n and $COST^*(n)$ is the minimal cost of a path from the initial state to a goal state via n. [TT entries for the nodes on the optimal path never overestimate C^*].
(P3) If $b < C^*$, then $b <$ DFSTT2$(root, b, [root])[1] \leq C^*$. [the cutoff bound b will increase as long as $b < C^*$].
(P4) If $C^* \leq b$, then DFSTT2$(root, b, [root])$ returns a path with cost $\leq b$. [if the cutoff bound b ever reaches C^*, the solution will be found].

The key property is P2, which guarantees that the TT behaves like an admissible heuristic. To prove P2, we consider how TT_i, the state of the transposition table at step i, changes as the algorithm progresses. Starting with an empty transposition table at step 0 ($TT_0 = \emptyset$), it can be shown straightforwardly by induction on time step i that P2 is correct. P2 applies to all entries that are in the TT at any given time – although replacement might cause entries to be deleted from the TT, it does not matter what the replacement policy is, because we only care that P2 holds true for all entries currently in the TT, and we do not care how or when entries are deleted. Given P1 and P2, properties P3 and P4 are straightforward properties of standard IDA*, so their proofs are omitted. The iterative deepening (Fig 1) starts with an initial cutoff of $bound_1 = h(root) \leq C^*$. By Properties 2,3, and 4, IDA*+DFSTT2 will iteratively increase the cutoff bound and search until an optimal path is found, at which point it will terminate. Thus, this algorithm is admissible.□

A different admissible approach was implemented in the RollingStone sokoban solver (Junghanns 1999). Assume (without loss of generality) unit edge costs. Instead of storing the lowest cost estimate found under an exhaustively searched node in the TT, this policy stores $bound - g(n) + 1$.[2] Rather than storing this value in the tree after searching the subtree, the value is stored in the TT *before* descending into the tree. Cycling back into this state will result in $g(s)$ being higher than its previous value, resulting in a cutoff (thus, this strategy does not require a separate cycle detection mechanism). It is easy to see that this RollingStone (**RS**) strategy is admissible:[3] Let g_1 and g_2 be g-values of n via paths p_1 and p_2, respectively. Assume RS first reaches n via p_1, and $(n, bound - g_1 + 1)$ has been stored in the TT. Suppose that we later encounter n via p_2. If $g_1 \leq g_2$, a cut off happens because $bound - g_1 + 1 + g_2 > bound$ (p_2 is longer than p_1).

[2]With non-unit edge costs, $bound - g(n) + \epsilon$ is stored, where ϵ the smallest edge weight in the graph.

[3]Since RS detects cycles using the TT, the replacement nodes must not replace nodes on the current search path - this is easily enforced.

```
function DFSTT2($n, bound, path$): (real,real)
 1: if $n$ is a goal state then
 2:     $solved := true$; $answer := path$; return $(0, 0)$;
 3: end if
 4: if successors($n$) $= \emptyset$ then
 5:     $new\_esti := \infty$; $new\_bound := \infty$;
 6: else
 7:     $new\_esti := \min\{\mathrm{ET}(m) | m \in$ successors($n$)$\}$;
 8:     $new\_bound := \min\{\mathrm{BD}(m) | m \in$ successors($n$)$\}$;
 9: end if
10: store $(n, new\_esti)$in $TT$;
11: return $(new\_esti, new\_bound)$;
```

Where ET(m) :=

Case 1: $c(n, m) +$ Lookup(m), if $path + m$ forms a cycle

Case 2: $c(n, m) +$ DFSTT2$(m, bound - c(n, m), path + m)[0]$, if $c(n, m) +$ Lookup$(m) \leq bound$

Case 3: $c(n, m) +$ Lookup(m), if $c(n, m) +$ Lookup$(m) > bound$

BD(m) :=

Case 1: ∞, if $path + m$ forms a cycle

Case 2: $c(n, m) +$ DFSTT2$(m, bound - c(n, m), path + m)[1]$, if $c(n, m) +$ Lookup$(m) \leq bound$

Case 3: $c(n, m) +$ Lookup(m), if $c(n, m) +$ Lookup$(m) > bound$

Figure 5: DFSTT2: An admissible algorithm

If $g_1 > g_2$, we reexpand n to try to find a solution within the current bound.

We now propose DFSTT2+RS, a hybrid strategy combining DFSTT2 and RS. Instead of storing (n, new_esti), we store $(n, max\{new\_esti, esti(n), bound - g(n) + \epsilon\})$ where ϵ is the smallest edge cost ($\epsilon = 1$ in our planning domains below). In this hybrid strategy, the value is stored after the search under node m is exhausted, while RS stores $bound - g(n) + \epsilon$ before descending into the tree. For all nodes, the TT entry for DFSTT2+RS dominates both DFSTT2 and RS, and it is easy to extend the proof of admissibility for DFSTT2 to show that DFSTT2+RS is admissible.

An alternate approach to addressing the corruption of finite TTs by cycles is to completely ignore cycles. This algorithm, DFSTTIC, is identical to DFSTT1 (Fig 3), except that the BD computation rule is the following: (case 1) BD(m) := $c(n, m)$ + DFSTTIC$(m, bound - c(n, m), path + m)$, if $c(n, m)$ + Lookup$(m) \leq bound$, and (case 2) BD$(m) := c(n, m) +$Lookup(m), if $c(n, m) +$ Lookup$(m) > bound$. Unlike the BD computation rule for DFSTT1, this modified rule lacks a cycle check. DFSTTIC is easily seen to *almost always* returns the optimal solution path in finite time. However, it can fail to terminate in graphs that contain a cycle of cost 0. Thus, DFSTTIC is not admissible.

4. Complete, Admissible IDA*+TT

In addition to admissibility, another important property is completeness, i.e., will the algorithm always terminate even

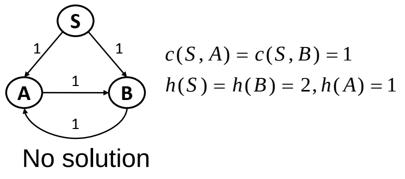


Figure 6: Counterexample for Proposition 4

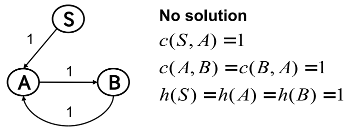


Figure 7: An example in which both DFSTT2 and DFSTT2+RS returns "no solution", but RS never terminates

when there is no solution?

Definition 2 *A search algorithm is* **complete** *if it is admissible, and returns* no solution *in finite time when no solution exists.*

Proposition 4 *IDA* using DFSTT2, RS, and DFSTT2+RS are incomplete.*

Proof: Consider the search graph in Figure 6.

First, consider DFSTT2: $bound$ is initially $h(S) = 2$. Due to the TT updates for $esti(A)$ and $esti(B)$, the search enters a pathological alternation where on the odd iterations, we generate the paths S, A, B, and S, B, and on the even iterations, we generate the paths S, B, A and S, A. On each iteration, $bound$ increases by 1. The TT entries are updated as follows: when $bound = i$, and $esti(A) = i, esti(B) = i - 1$ at the beginning of the iteration, then $esti(A) = i, esti(B) = i + 1$ at the end of the iteration; when $bound = i$, and $esti(A) = i - 1, esti(B) = i$ at the beginning of the iteration, then $esti(A) = i+1, esti(B) = i$ at the end of the iteration. This process alternates forever and IDA*+DFSTT2 never terminates. IDA*+RS also fails on this graph because the bound never stops increasing. □

Note that while RS fails to terminate on all instances which have this type of cycle, DFSTT2 is able to solve a subset of these instances, i.e., DFSTT2 dominates RS. Figure 7 illustrates this.

We now propose DFSTT3 (Fig 8), a modified version of DFSTT2 which is both complete and admissible. The main idea is to store not only $esti$, but also the g-cost associated with $esti$. When we revisit a node, the g-cost information allows us to determine whether we are revisiting a node that has already been reached via a shorter path. Lookup2 is similar to Lookup, except that it retrieves (and stores, if no entry was present) the g-cost in addition to $esti$. Additionally, $esti(n)$ is passed as an argument of DFSTT3 to be used as a conservative estimation when a cycle is detected. This allows us to label such nodes as dead ends. In the search graph in Figure 6, the algorithm labels both A and B as dead ends, and correctly returns ∞ as the bound.

```
function DFSTT3($n, bound, path, esti$) : (real,real)
 1: if $n$ is a goal state then
 2:     $solved := true$; $answer := path$; return $(0, 0)$;
 3: end if
 4: if successors($n$) $= \emptyset$ then
 5:     $new\_esti := \infty$; $new\_bound := \infty$;
 6: else
 7:     $new\_esti := \min\{\text{ET}(m) | m \in \text{successors}(n)\}$;
 8:     $new\_bound := \min\{\text{BD}(m) | m \in \text{successors}(n)\}$;
 9: end if
10: store $(n, new\_esti, g(path))$ in $TT$;
11: return $(new\_esti, new\_bound)$;
```

Where: ET(m) :=

Case 1: $esti$, if $path + m$ is suboptimal

Case 2: $c(n,m)$ + DFSTT3$(m, bound - c(n,m), path + m, \text{Lookup2}(m))[0]$,
if $c(n,m) + \text{Lookup2}(m) \leq bound$

Case 3: $c(n,m) + \text{Lookup2}(m)$,
if $c(n,m) + \text{Lookup2}(m) > bound$

BD(m) :=

Case 1: ∞, if $path + m$ is suboptimal

Case 2: $c(n,m)$ + DFSTT3$(m, bound - c(n,m), path + m, \text{Lookup2}(m))[1]$,
if $c(n,m) + \text{Lookup2}(m) \leq bound$

Case 3: $c(n,m) + \text{Lookup2}(m)$,
if $c(n,m) + \text{Lookup2}(m) > bound$

* $path + m$ is suboptimal iff $path + m$ forms a cycle, or TT contains an entry $(m, esti_m, g(m))$, and $g(m) < g(path + m)$.

Figure 8: DFSTT3: An admissible and complete algorithm

Theorem 2 *Given an admissible heuristic function, IDA*+DFSTT3 is complete.*

Proof sketch: Properties analogous to P1-P4 for DFSTT2 are easily seen to be true for DFSTT3, so it follows that DFSTT3 is admissible.

To show completeness: Let MAX $:= \max_n\{g_{max}(n) + h(n)\}$, the longest acyclic path in the graph (assuming wlog $h(n) < \infty$). The following properties hold:
(P5) If the transposition table contains an entry $(n, esti, g(n))$ for node n, $esti \leq \text{MAX} - g(n)$. This can be proven straightforwardly by induction, similar to (P2).
(P6) If no solution exists, DFSTT3$(root, \text{MAX}, [root])[1] = \infty$. This can be shown by showing that when computing BD(n), either $g(n) + \text{esti}(n) \leq \text{MAX}$, or the current path is suboptimal. Thus, if there is no solution within cost MAX, then for every node in this last iteration the computed value of $BD(n) = \infty$. □

Like DFSTT2, DFSTT3 can be combined with RS. Let $p = (esti(n), g_{TT}(n))$ be a pair of $esti$ and g-value currently in the TT, and $g(n)$ be the g-value of the current path, and $v = max\{new\_esti, bound - g(n)\}$. This hybrid, DFSTT3+RS, stores $(n, v, g(n))$ in the TT if $v > esti(n)$. Otherwise, p is preserved. Note that the second term of v is $bound - g(n)$ instead of $bound - g(n) + \epsilon$ and that is important considering which g-value is stored to preserve P5 in Theorem 2. As the experimental results show, this difference of ϵ results in a large performance degradation compared to RS; however, in order to maintain completeness, we have not yet been able to increase the second term.

5. Replacement Policies

So far, we have identified a set of transposition table update strategies which robustly guarantees the admissibility of IDA*+TT. We now describe several TT replacement policies that we have implemented. Since our analysis of DFSTT2 and DFSTT3 above made no assumptions about replacement policy, all of the replacement policies below can be safely used without compromising the admissibility of these algorithms.

A trivial policy is *no replacement* – add entries until the table is full, but entries are never replaced (although the stored estimated values for the cached nodes will be updated as described above). *Stochastic Node Caching* (SNC) is a policy based on (Miura and Ishida 1998), which seeks to only cache the most commonly revisited nodes in memory by probabilistically storing the state with some constant probability p. After the table is full, there is no replacement.

The standard practice for TT replacement 2-player games is *collision-based replacement*, which decides to either replace or retain the entry where a hash collision for a table entry occurs. The most common collision resolution policy keeps the value associated with the deeper search (which has a larger subtree size, and presumably saves more work).

An alternative to collision-based replacement is *batch replacement*, which has also been studied in two-player games (Nagai 1999). This is similar to garbage collection, and is triggered by running out of space. In this scheme, memory management for the TT is done using a dedicated object memory pool – there is a pool (linked list) of TT entry objects which are initially allocated and empty. When a new TT entry object is requested, the first available element from the pool is returned; when a TT entry is "freed", the object is marked and returned to the pool.

When the TT becomes full, the nodes are sorted based on one of the replacement criteria: (a) subtree size (prefer larger subtrees since they tend to save the most computation), (b) backed up cost estimate for the node (prefer the most promising nodes), and (c) the number of accesses for the entry (prefer frequently accessed entries). Then, the bottom $R\%$ of the entries are chosen and marked as "available". These entries are not immediately discarded – batch replacement merely designates the set of entries which will be overwritten (with equal priority) as new nodes are generated, so the entries remain accessible until overwritten.

6. Experimental Results

We implemented the IDA* variants described in this paper as a replacement search algorithm for a recent version of the Fast-Downward domain-independent planner using abstraction heuristics (Helmert, Haslum, and Hoffmann 2007), and evaluated their performance. The following TT replacement strategies were considered: (a) no replacement, (b) stochastic caching, (c) collision-triggered replacement based

Algorithm	TT Replacement Policy	Num Solved	Tot. Runtime (seconds)
A*		173	539
DFS	No TT	128	178098
DFSTT2	TT, No Replace	183	73477
DFSTT2	Replace 0.3, subtree size	194	52256
RS	Replace 0.3, subtree size	195	68319
DFSTT2+RS	TT, No Replace	189	106147
DFSTT2+RS	Stochastic Caching, p=0.001	187	213765
DFSTT2+RS	Replace 0.3, est	194	40290
DFSTT2+RS	**Replace 0.3, subtree size**	**195**	**66960**
DFSTT2+RS	Replace 0.3, access freq.	194	39187
DFSTT2+RS	Collision, est	189	141249
DFSTT2+RS	Collision, subtree size	192	114057
DFSTT3+RS	Replace 0.3, subtree size	152	170296

Table 1: Performance on 204 IPC planning instances, 2GB memory total for solver, 10 hours/instance. **Runtimes include successful runs only**.

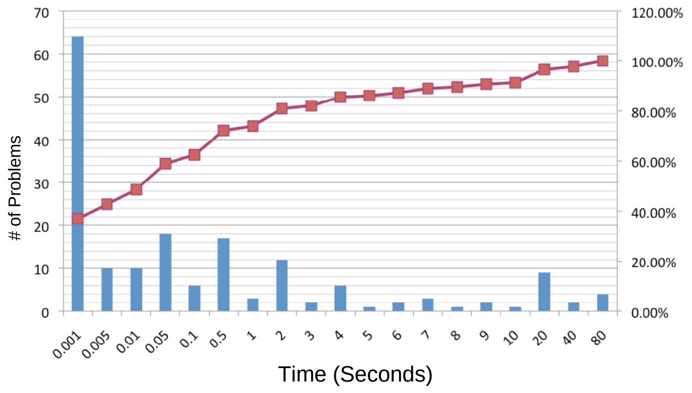


Figure 9: Runtime distribution of A* (solved instances only)

on subtree size and estimated cost (d) batch replacement based on subtree size, estimated cost, and access frequency.

The fraction of nodes marked as available by batch replacement was $R = 30\%$. The Fast Downward abstraction size was 1000 for all configurations. The algorithms were tested on a set of 204 instances from the IPC planning competition (IPC3: depots, driverlog, freecell, zenotravel, rovers, satellite; IPC4: pipes tankage, pipes no tankage, airport, psr small; IPC6: sokoban, pegsol).[4] Each algorithm was allocated 10 hours/instance. The experiments (single-threaded) were run on a 2.4GHz Opteron. 2GB RAM was allocated for the entire solver (including the transposition table and abstraction table) – the TT is automatically sized to fully use available memory, depending on the type of information needed by the TT replacement policy (i.e., $esti$, g-value, and auxiliary data used by the replacement policy). The results are shown in Table 1.

First, we compared the performance of our IDA* variants with the performance of the default A* search algorithm in Fast-Downward (Helmert, Haslum, and Hoffmann 2007). A* solved 173 problems, but exhausted memory on the remaining problems The total runtime (540 seconds) for A* is very low compared to the IDA* variants because the search terminates when memory is exhausted. Note that the DFS+TT variants also solve these easy problems quickly. DFSTT2+RS solves 182 problems (9 more than A*) within 30 minutes (total); the remainder of the 66960 seconds were spent on the most difficult instances which were solved by DFSTT2+RS (but not solved by A*). Figures 9 and 10 show the runtime distributions of A* and DFSTT2+RS, respectively, on the set of planning benchmarks (problems which were not solved are excluded). We also took the 173 problems which were solved by both A* and DFSTT2+RS, and computed the ratio of the runtimes of these two algorithms, $time(DFSTT2 + RS)/time(A*)$ for each problem. The distribution of these runtime ratios is shown in Figure 11. This shows that DFSTT2+RS was 1-2x slower than A* on 44 instances, 2-3x slower than A* on 61 instances, and so on. Although IDA* is much slower than A* on a fraction of the benchmarks, its runtime was within a factor of 4 of A* on most (76%) of the instances. Since IDA* can solve problems which A* fail on due to memory exhaustion, the slowdown on the easier instances seems to be worthwhile tradeoff.

It is clear that using a transposition table results in a significant improvement over plain IDA*. Among the IDA*+TT variants, the DFSTT2+RS hybrid strategy with subtree size based replacement resulted in the best overall performance.

The batch replacement-marking methods significantly outperformed IDA*+TT without replacement and SNC, showing the importance of replacement. Interestingly, some of the variants using collision-based replacement performed worse than no replacement, showing that choice of replacement policy is critical for performance.

The DFSTT3 strategy performed poorly compared to DFSTT2 and RS, showing that there is a significant price to be paid when we store conservative values in the TT in order to guarantee completeness. An evaluation of DFSTT3 with unsolvable problems is future work.

7. Related Work

Transposition tables are used extensively in 2-player games. In 2-player games, the problem of incorrect results caused by storing path-dependent results in the transposition table has been studied as the Graph-History Interaction (GHI) problem (Campbell 1985; Kishimoto and Müller 2004). As with the single-agent case investigated in this paper, handling the GHI imposes some overhead. Since the ultimate goal in 2-player games is usually strong play, many implementations trade off some correctness (of the backed-up evaluations) in order to achieve higher speed. However, handling the GHI is still important even in 2-player games

[4]To avoid wasting a lot of time on problems that couldn't be solved by any configuration, our benchmark set was selected from these problem sets based on preliminary experiments.

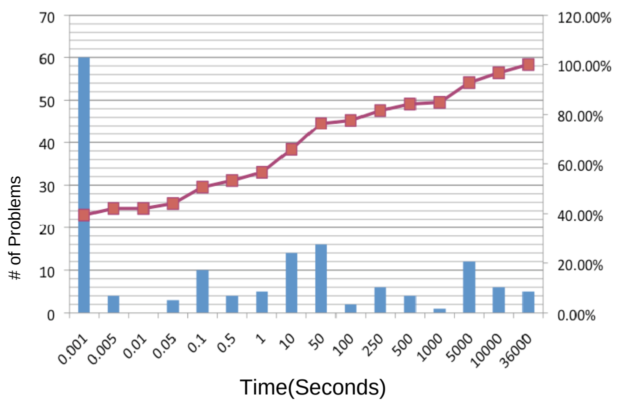


Figure 10: Runtime distribution of IDA*+DFSTT2TT (solved instances only)

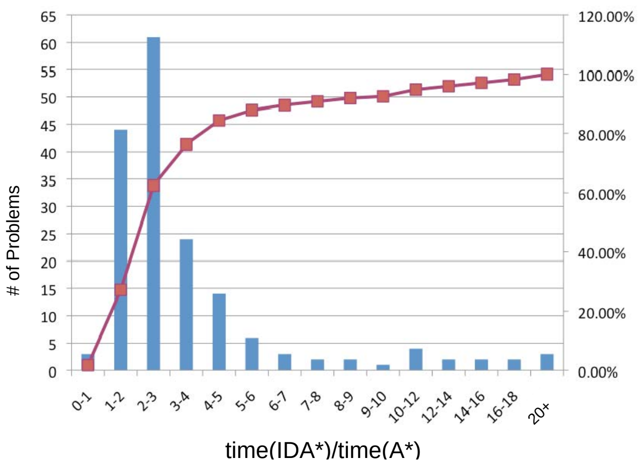


Figure 11: IDA*+DFSTT2+RS vs. A* performance

especially in solving difficult game positions (Nagai 2002; Schaeffer et al. 2007).

Although several previous papers have considered transposition tables for single-agent search (Reinefeld and Marsland 1994; Junghanns and Schaeffer 2001), the use of transposition tables is not as widespread as in 2-player games. As far as we know, this is the first paper to focus on TT usage and the issues that arise when integrating TT, cycle detection, and TT replacement policies for single-agent search. Also, this is the first paper to give formal analysis for an IDA*+TT variant which guarantees that any replacement strategy can be used (with consistent or inconsistent heuristics) without sacrificing admissibility.

An alternate approach to duplicate detection uses a finite-state machine (FSM) to detect sequences of operators that generate duplicates (Taylor and Korf 1993). This approach uses very little memory during search; however, a separate preprocessing phase is required for learning the FSM.

MREC (Sen and Bagchi 1989) behaves similarly to A* while memory is available. Once memory is filled up, it executes IDA* from the fringe nodes. The set of states stored in memory by MREC remains static (no replacement of states occurs). SNC (Miura and Ishida 1998) is similar to MREC, except that when memory is available, SNC probabilistically caches nodes with probability p. After memory runs out, SNC behaves like MREC.[5]

While MREC and SNC never update the set of nodes that are cached in memory, MA* (Chakrabarti et al. 1989) and a simplified, improved variant, SMA* (Russell 1992) seek to dynamically update the set of nodes that are in limited memory. SMA* behaves like A* until memory is full. At this point, SMA* removes the shallowest, highest-f-cost node from OPEN (freeing up space for another node), and backs up its value to its parent. The parent keeps the minimum of its children's values, and is placed back on the OPEN list. This keeps the most promising portions of the search space in memory. Keeping the backed-up values of the less promising nodes ensures that the algorithm can regenerate the forgotten portions in order of how promising they appeared at the time they were forgotten. ITS (Ghosh, Mahanti, and Nau 1994) is a related approach with a tree search formulation. Our work differs from this previous line of work in two respects. MREC, SNC, SMA* are based on A*, rather than IDA*. Second, the previous work on MREC, SNC, SMA* and ITS propose and analyze a particular search algorithm with a specific policy by which nodes are stored (and possibly removed from) memory. In contrast, while Theorems 1 (admissibility) and 2 (completeness) consider a particular search strategy (IDA*+TT), we make minimal assumptions about the TT replacement policy, which determines which nodes are retained in memory.

Recent work has identified several ways to address memory limitations in search, including reducing the set of nodes which need to be stored (Korf et al. 2005), as well as increasing memory capacity by using external storage (Korf 2008). These methods can be applied to domain independent planning (Edelkamp and Jabbar 2006; Zhou and Hansen 2007). Another approach is to use massive amounts of aggregate memory in a distributed search algorithm (Kishimoto, Fukunaga, and Botea 2009). However, although these methods allow problems to be solved which would be otherwise unsolvable with previous algorithms, they eventually run out of memory and terminate on harder problems. In contrast, IDA*+TT will, in principle, continue running until the search is finished (although the IDA*-based search may not be as efficient as these methods).

[5]While our experiments considered replacement strategies similar to MREC (TT/no replacement) and SNC (TT+SNC), these are not intended to substitute for a direct comparison (future work).

8. Discussion and Future Work

In this paper, we investigated transposition tables for IDA*. Our main contributions are:

- Theoretical analysis showing that straightforward implementations of transposition tables can result in inadmissible and incomplete behavior. This is the result of subtle interactions between the TT and cycle detection, particularly when TT capacity is finite. It turns out that replacement behavior can affect the correctness of the TT entries;
- Identification of several strategies which are admissible, regardless of replacement policy;
- An admissible and complete IDA*+TT algorithm; and
- An experimental study which showed that IDA*+TT can significantly improve performance on a recent version of the state-of-the-art Fast Downward domain-independent planner, which uses A*.

A key advantage of IDA*+TT over A* is that while A* terminates when memory is exhausted, IDA*+TT will use all available memory and continue searching for a solution until one is found or time runs out. In applications such as domain-independent planning, this is important because it is difficult to estimate the amount of memory required for an arbitrary planning instance a priori. One previous advantage of A* over IDA* was completeness – our IDA*+DFSTT3 algorithm is both admissible and complete, but the conservative rules for backing up values resulted in relatively poor performance. Improving the performance of complete IDA*+TT is an area for future work.

Our experimental results indicate that the replacement policy can have a significant impact on performance. The effectiveness of TT replacement policies may depend on the problem, as well as the heuristic, and this remains an area for future work. However, since we have shown that a relatively simple modification to IDA*+TT results in an algorithm which is admissible regardless of replacement policy, the correctness of IDA*+TT and the choice of replacement policy has been modularized – a wide range of replacement policies can now be investigated further without being concerned about compromising admissibility (and for DFSTT3, completeness as well).

9. Acknowledgments

This research was supported by JST, JSPS, and the MEXT CompView GCOE. Thanks to the anonymous reviewers for helpful comments.

References

Campbell, M. 1985. The graph-history interaction: On ignoring position history. In *1985 ACM Annual Conference*, 278–280.

Chakrabarti, P.; Ghose, S.; Acharya, A.; and de Sarkar, S. 1989. Heuristic search in restricted memory. *Artificial Intelligence* 41(2):197–221.

Edelkamp, S., and Jabbar, S. 2006. Cost-optimal external planning. 821–826.

Ghosh, S.; Mahanti, A.; and Nau, D. 1994. ITS: An efficient limited-memory heuristic tree search algorithm. In *Proc. AAAI*, 1353–1358.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4* 4(2):100–107.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS-07*, 176–183.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence* 129:219–251.

Junghanns, A. 1999. *Pushing the limits: new developments in single-agent search*. Ph.D. Dissertation, University of Alberta.

Kishimoto, A., and Müller, M. 2004. A general solution to the graph history interaction problem. In *Proc. AAAI*, 644–649.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proc. ICAPS*, 201–208.

Korf, R.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

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

Korf, R. 2008. Linear-time disk-based implicit graph search. *Journal of the ACM* 55(6).

Miura, T., and Ishida, T. 1998. Stochastic node caching for memory-bounded search. In *Proc. AAAI*, 450–456.

Nagai, A. 1999. A new depth-first search algorithm for AND/OR trees. Master's thesis, Univ. of Tokyo.

Nagai, A. 2002. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. Ph.D. Dissertation, University of Tokyo.

Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Russell, S. 1992. Efficient memory-bounded search methods. In *Proc. ECAI*.

Schaeffer, J.; Burch, N.; Björnsson, Y.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2007. Checkers is solved. *Science* 317(5844):1518–1522.

Sen, A., and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proc. IJCAI*, 297–302.

Taylor, L., and Korf, R. 1993. Pruning duplicate nodes in depth-first search. In *Proc. AAAI-91*, 756–761.

Zhou, R., and Hansen, E. 2007. Parallel structured duplicate detection. 1217–1223.

Zobrist, A. 1970. A new hashing method with applications for game playing. Dept. of CS, Univ. Wisconsin-Madison, Tech Rept.