2E1-2

IDA*探索におけるトランスポジションテーブルについて

On Transposition Tables for IDA*

赤木 維磨[*1] 岸本 章宏[*1*2] Alex Fukunaga [*3]
Yuima Akagi Akihiro Kishimoto

[*1]東京工業大学大学院 情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

[*2]科学技術振興機構 さきがけ
JST PRESTO

[*3]東京大学大学院総合文化研究科
Graduate School of Arts and Sciences, University of Tokyo

Transposition tables are a well-known method for pruning duplicates in heuristic search. This paper summarizes recent results we have obtained regarding transposition tables for IDA*. We show that some straightforward implementations of IDA* with transposition tables (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 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* [3] 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* [6]. 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 [9] 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.

This paper summarizes the results of our recent study of transposition tables for IDA*. We first describe analytical results showing 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). We identify IDA*+TT algorithms which are guaranteed to find the optimal solution (regardless of replacement policy), but are incomplete. We also describe an IDA*+TT algorithm which is complete. Then, we discuss replacement policies for transposition tables, and empirically demonstrate the effectiveness of IDA*+TT in domain-independent planning. Due to space, this paper only presents the main results. For proofs and more details, see [1, 2].

連絡先: 岸本章宏 東京工業大学大学院 情報理工学研究科 数理・計算機科学専攻 152-8550 東京都目黒区大岡山 2-12-1-W8-25 (Email:kishimoto@is.titech.ac.jp)

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

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

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 [6].

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 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, 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

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 $c(n, m) + h(m) > bound$

図 2: DFS for standard IDA*

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* **exact** *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 exact 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

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 then 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 exact.*

1 For example, the 15-puzzle implementation in [9] 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**

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

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 exact (for some replacement policies).*

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 exact regardless of TT replacement policy.

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

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

3. Exact 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 4. 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 exact.*

```
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) = ∅ then
 5:     new_esti := ∞; new_bound := ∞;
 6: else
 7:     new_esti := min{ET(m)|m ∈ successors(n)};
 8:     new_bound := min{BD(m)|m ∈ 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) ≤ 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) ≤ bound
Case 3: c(n, m) + Lookup(m),
        if c(n, m) + Lookup(m) > bound
```

図 4: DFSTT2: An exact algorithm

A different exact approach was implemented in the RollingStone sokoban solver [5]. 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 exact:*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). 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 exactness for DFSTT2 to show that DFSTT2+RS is exact.

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 exact.

3.1 Complete, Exact IDA*+TT

In addition to exactness, another important property is completeness:

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

None of the IDA*+TT variants described above is complete. We have developed DFSTT3, which is a complete, IDA*+TT algorithm based on DFSTT2. 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. The lookup function 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 the recursive DFSTT3 function to be used as a conservative estimation when a cycle is detected. This allows us to label such nodes as dead ends. As with DFSTT2, this complete algorithm can be combined with RS. Details and proof of completeness are in [1]

4. Replacement Policies

So far, we have identified a set of transposition table update strategies which robustly guarantees the exactness of IDA*+TT. We now describe several TT replacement policies that we have implemented. Since our analysis of DFSTT2 above made no assumptions about replacement policy, all of the replacement policies below can be safely used without compromising the exactness 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 [7], 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 [8]. 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 an 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

∗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.

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

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

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.

5. 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 [4], and evaluated their performance. The following TT replacement strategies were considered: (a) no replacement, (b) stochastic caching, (c) collision-triggered replacement based 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 2.8GHz Xeon. 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.

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*).

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.

6. Conclusions

Our experimental results show that TT can significantly improve the performance of IDA*, and that replacement policy has a significant impact on IDA*+TT performance. While our theoretical results show that in general using arbitrary replacement strategies with straightforward implementations of IDA*+TT can result in suboptimal solutions being returned, we have shown that there exist strategies such as DFSTT2 which are provably exact, regardless of replacement strategies, allowing us to safely apply the most effective replacement policies.

参考文献

[1] Y. Akagi. IDA* using transposition tables and its application to planning. Master's thesis, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2010.

[2] Y. Akagi, A. Kishimoto, and A. Fukunaga. On transposition tables for single-agent search and planning: Summary of results. In *submitted*, 2010.

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

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

[5] A. Junghanns. *Pushing the limits: new developments in single-agent search*. PhD thesis, University of Alberta, 1999.

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

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

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

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

∗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.