

# A depth-first approach to target-value search

Tim Schmidt, Lukas Kuhn, Bob Price, Johan de Kleer, Rong Zhou

Palo Alto Research Center  
3333 Coyote Hill Rd  
Palo Alto, CA-94304

## Abstract

In this paper, we consider how to improve the scalability and efficiency of target-value path search on directed acyclic graphs. To this end, we introduce a depth-first heuristic search algorithm and a dynamic-programming method to compute the heuristic's pattern database in linear (in the number of edges) time. We show the benefits of the new approach over previous work on this problem (Kuhn et al. 2008a).

## Introduction

In a target value-path problem, we are interested in finding a path between two nodes in a graph, such that some additive function (typically the sum) of the path's edge weights or values comes as close as possible to the target-value. Such problems arise for example when integrating model-based planning and diagnosis (Kuhn et al. 2008b). This work exploits degrees of freedom in attaining production for maximising information gain about the production system in order to increase long-run productivity. Intuitively the graph models valid action sequences for attaining some production goal and the edge weights represent the diagnosis engine's confidence in the operability of the respective component. Assuming single, non-intermittent faults, selecting the path whose predicted success probability is as close as possible to 0.5 will maximise the diagnostic engine's information gain about the system's true state (Liu et al. 2008). Other potential domains include comprehensive training programmes, with complex temporal and causal interdependencies between courses where participants need to reach certain point thresholds (i.e. university studies or mandatory professional training programmes) or determining nightly-build processes out of a large set of interdependent transformation (compilation, automated refactorings, code generation, etc.) and analysis tasks (unit and integration tests, code coverage, model checking, clone detection, profiling, etc.) to make best use of allotted time.

## Problem definition

Given a directed acyclic graph  $G = (V, E)$  with non-negative edge values, a *target-value-path* (or *tvp* in short)

Copyright ©2009, Palo Alto Research Center (www.parc.com), All rights reserved

between two vertices  $v_0, v_g \in V$  with *target-value*  $tv$  is some path between  $v_0$  and  $v_g$ , whose value is as close as possible to  $tv$ . We define the value  $g(p)$  of a path  $p$  as the sum of its edge values. Let  $P_{v_0, v_g}$  be the set of all paths between  $v_0$  and  $v_g$  in  $G$ . Then we define  $P_{v_0, v_g}^{tv} = \operatorname{argmin}_{P_{v_0, v_g}} (|tv - g(p)|)$

the set of paths between  $v_0$  and  $v_g$  with minimal deviation from the target-value as the *target-value path set* with respect to  $v_0, v_g, tv$ . In the following *target-value search* (or *tvs*, in short) refers to a mapping of tuples  $(v_0, v_g, tv)$  to some element of  $P_{v_0, v_g}^{tv}$ .

## Conventions

For reasons of clarity and brevity, we limit our discussion to *connection graphs*. The connection graph  $C_{v_0, v_g}$  is the subgraph of  $G$  containing  $v_0, v_g$  and those vertices in  $V$  that are both descendants of  $v_0$  and ancestors of  $v_g$  as well as all edges ( $\in E$ ) between them. We note, that  $C$  can be extracted by creating the intersection of a breadth-first sweep from  $v_0$  along successor links and from  $v_g$  along predecessor links, in time and space linear in the size of  $G$  (hence  $O(|V| + |E|)$ ). We generally assume that predecessors can be accessed efficiently and that edge values are positive.

In terms of notation, we omit indices where they are implied by context. We use the term *prefix* for any path from  $v_0$  to some vertex in  $C$ , the term *suffix* for any path from some vertex or interchangeably from (the last vertex of) some prefix to  $v_g$ ; note that any vertex or prefix will have at least one suffix. The term *completion* of a prefix denotes its concatenation with any of its suffixes and the term *optimal completion* of a prefix w.r.t a target-value, denotes the completion that is closest to the target-value.

*Target-value search* is a challenging problem because it does not exhibit the property of *optimal substructure*, a prerequisite for *greedy* or *dynamic programming* approaches (as are often used in shortest-path problems, for example). While any decomposition of a target-value path  $tvp = tvp_{pre} \circ tvp_{suf}$  w.r.t a target-value  $tv$  can be formulated as target-value paths w.r.t  $tv_{pre}$  and  $tv_{suf}$ , the target-values are interdependent (except for trivial decompositions like  $\epsilon \circ tvp$ ) as  $tv_{pre} = tv - g(tvp_{suf})$  and  $tv_{suf} = tv - g(tvp_{pre})$  and so are the respective *cost functions* for the subproblems. See figure 1 for an example. In the worst case, all prefixes in  $C$  up to (roughly) the target-value will have to be gen-

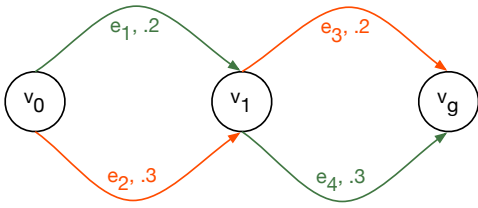


Figure 1: *tv*s does not exhibit *optimal substructure*. Consider the above graph for  $tv = 5$ . After we expanded  $v_0$ , we have two paths  $\langle e_1 \rangle, \langle e_2 \rangle$  to  $v_1$ . Both can lead to optimal solutions with the right completion (i.e.  $\langle e_1, e_4 \rangle$  and  $\langle e_2, e_3 \rangle$ ), the selection of which depends on the whole prefix, not only on its last vertex.

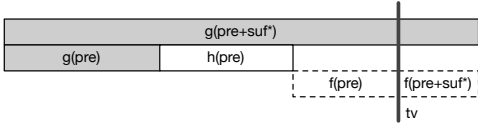


Figure 2: an example of why a heuristic that underestimates suffix lengths will not lead to an admissible *tv*s heuristic: here  $f(pre) = |tv - (g(pre) + h(pre))| > f(pre + su f^*) = |tv - g(pre + su f^*)|$  while  $g(pre) + h(pre) < g(pre + su f^*)$ .

erated during a *tv*s This leads us to believe that *tv*s is in *EXPTIME*, as the number of these prefixes can be exponential in the number of vertices of  $C$ .

### Heuristic Target Value Search

A straightforward approach to tackle target-value search problems is to use some estimate  $h$  of suffix lengths and search through path space with  $A^*$  (Hart, Nilsson, and Raphael 1968) using an *inadmissible* guiding function, such as  $f(pre) = |g(pre) + h(pre) - tv|$ . Note that if  $h$  underestimates suffix lengths (e.g. it is admissible for shortest-path search),  $f$  will generally not be admissible due to the *non-linearity* introduced by the absolute value operator. See figure 2 for an example. The basic idea is to find a good solution  $tv_{p_{cur}}$  quickly and use it to prune the Open List of all prefixes whose  $g$  value exceeds  $tv + |tv - g(tv_{p_{cur}})|$ . The search terminates if either a perfect solution is found (i.e.  $g(tv_p) = tv$ ) or the Open List is empty (returning  $tv_{p_{cur}}$ ).

### Previous Work

(Dow and Korf 2007) show how an admissible heuristic can be constructed for the non-additive, max-edge cost function of the *treewidth* problem and then be employed in best-first search. (Kuhn et al. 2008a) construct a *pattern database* (Culberson and Schaeffer 1998) to derive a *consistent heuristic* for best-first target value search. They show that problem structure can be leveraged in two ways. First, prefixes ending in the same vertex and having equal value can be considered duplicates and be pruned from the search tree. Second, given the pattern database for the graph, one can both use it to guide the search and detect when the problem of finding an optimal suffix for some prefix degenerates into a shortest or longest path problem, which is

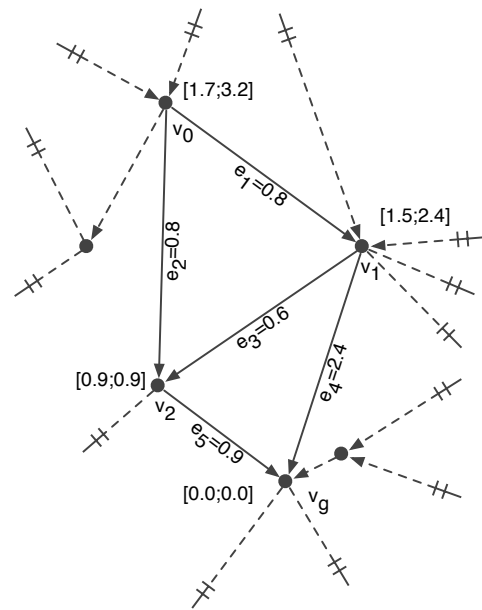


Figure 3: the connection graph (solid edges) of  $v_0$  and  $v_g$  with edge values and entries of a single interval *pattern database*.

then straightforward to solve in vertex space. The pattern database (*pdb* in the following) maps each vertex to an interval representing the range of its completion lengths. It can be used to efficiently determine a lower bound of the deviation of the optimal completion of any prefix from the target-value as follows: given a target-value  $tv$  and the length of the prefix  $g(pre)$ ,  $tv_{suf} = tv - g(pre)$  is compared to the interval of  $pre$ 's last vertex. If  $tv_{suf}$  falls inside the interval, it may be possible to find a completion with exactly the target-value and the lower bound is hence 0. Otherwise, the distance to the interval is a lower bound of the deviation of the best completion for  $pre$ . In this case, the relative position of  $tv_{suf}$  to the interval reveals whether we can find the best completion through a *shortest-path* search (if  $tv_{suf}$  is greater) or a *longest-path* search (if  $tv_{suf}$  is less). Both can be solved efficiently (in a DAG) using dynamic programming, and corresponding admissible heuristics can be derived straightforwardly from the pattern database.

The  $f(pre)$  function is defined as 0 if  $tv - g(pre)$  falls inside the interval, otherwise as the distance of  $tv - g(pre)$  to the closest bound. This can be used in a fairly standard  $A^*$  search with duplicate detection as sketched out above. In contrast to the first approach, which in many cases (if there is no perfect  $tv_p$  in the graph) has to generate all prefixes in  $C$ , with  $g < tv + f^*$ , this approach can often make do with a small subset, typically offsetting the cost for constructing the database (especially if it can be reused, i.e. multiple queries are performed with the *same*  $v_g$ ). In the worst case, both algorithms have to generate all prefixes with values  $\leq tv + f^*$  in  $C$ , situating them in *EXPSPACE*. Also the algorithm for computing the *pdb* as given in (Kuhn et al. 2008a) has a worst-case exponential runtime complexity.

In the following, we will show how to extend the above

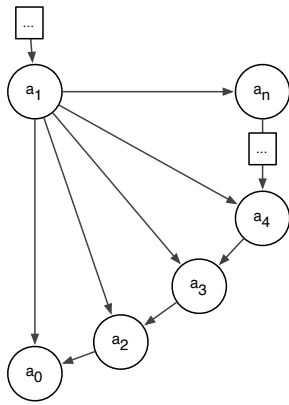


Figure 4: To compute both *shortest* and *longest* paths for its *pdb* (Kuhn et al. 2008a) proposes a backward propagation scheme that can require exponential time on graphs like this one.

pattern database concept to allow for tighter bounds and construction in time linear in the size of the connection graph. In the second part we will adapt *depth-first branch and bound search* to target-value problems, and finally give empirical evidence that the combination of these techniques allows us to scale target-value search to much larger problems than before.

### Multi-interval Pattern Databases

The database of (Kuhn et al. 2008a) stores a single interval per vertex, with the bounds comprising the least and largest value of that vertex’s suffixes. Thus the interval approximates the range of possible suffix values for that vertex (i.e. the respective values of the *shortest* and *longest* paths from the vertex to  $v_g$ ). We extend that concept, by allowing multiple, disjoint intervals per vertex, providing a finer-grained approximation. First, we give a short recapitulation of how Kuhn et al.’s database is constructed: Initially, all *pdb* entries are initialised to  $[+\infty; -\infty]$  (the largest/least possible lengths of shortest/longest paths from that vertex to  $v_g$ ), except  $v_g$ , which is set to  $[0; 0]$  and  $v_g$  is added to a (fifo) queue. Then, in each iteration, the next vertex is removed from the queue, and for each of its in-edges  $e$ , the lower bound of the predecessor is set to the *min* of its former lower bound and  $value(e)$  plus the lower bound of  $v$ . The same is done for the upper bound (using *max*). If the predecessor’s interval changes in the process, it is added to the queue. The algorithm terminates once the queue is empty.

This construction procedure is one of the two major culprits preventing the scaling of the algorithm to larger problems (see empiric evaluation). Consider the example in figure 4 under the following assumptions:  $a_0$  is currently head of the queue, vertices are returned by the ancestor function in the order of their numbering and each update changes the pattern database entry of the respective vertex. First  $a_0$  is expanded, thereby updating  $a_1$  and  $a_2$  and placing them on the queue, in that order. Now  $a_1$  is expanded, and all its predecessors (not shown in the figure) are placed on the queue. Next comes  $a_2$ . According to our assumptions,  $a_1$  gets up-

dated and is thus placed on the queue, along with  $a_3$ . Now all the predecessor of  $a_1$  are processed, until we again encounter  $a_1$  on the queue, resulting in its updated predecessors being added to the queue yet again. And so forth. In the worst case, each vertex is updated once for each distinct path between it and the goal vertex, resulting in exponential (in the number of vertices) worst-case running time.

This can be avoided by using a dynamic programming approach, as each database entry only depends on the entries of its successors in  $C$ . As we are dealing with acyclic graphs, processing vertices in some (inverse) topological order during construction ensures that all successor entries are already available. During construction, we store a counter with each vertex, initialised with its number of successors in the connection graph  $C$  (technically we only need to create/initialise the counter when we visit the vertex for the first time and can remove it, once it hits zero). We begin with a queue holding the  $v_g$  vertex, its *pdb* entry set to  $[0; 0]$ . At each step, we remove the first vertex from the queue, combine the intervals from all its successors in  $C$ , and decrement the successor counter of all its predecessors by one. Should the counter reach 0, we add the predecessor to the queue. The successor intervals are conceptually combined in three steps. At first, for each successor all intervals are transformed, by adding the corresponding edge value to all its bounds. Then, any pair-wise overlap within the union of transformed successor intervals is removed by replacing the culprits with the smallest covering interval. And last, while the number of intervals exceeds a user-defined maximum, the two closest intervals are fused. See figure 5 for an example.

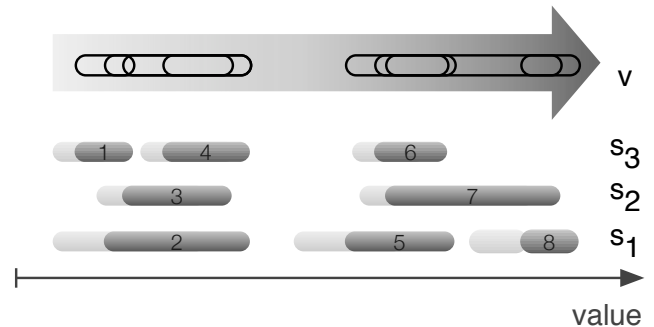


Figure 5: *pdb* entry construction for some vertex  $v$  with successors  $s_1, s_2, s_3$ : successor ranges are shifted by the value of the respective connecting edge. These intervals are accessed in ascending order of their lower bounds (denoted by numeric indices). We begin by fetching the first interval and set it as temp. While there are any unprocessed successor intervals, we fetch the next and compare it to temp. If they intersect we set temp’s upper-bound to next’s, else we add temp to  $v$ ’s *pdb* entry and set next as temp. Finally we add temp to  $v$ ’s entry, creating the entry shown in the upper row of this example.

Using this technique, each vertex in the connection graph will be processed only once. This can be shown through an induction proof: if all of the descendants of  $v$  have been accessed only once, then this is certainly true for its succes-

sors, so  $v$ 's successor counter will be 0, so  $v$  will be added to the queue and thus be processed once (induction step).  $v_g$  starts on the queue and has no descendants in  $C$ , so  $v_g$  (the global descendant) will only be accessed once (base case). From this it follows that each edge will be accessed twice (in the predecessor direction to update the counts and, in the other direction to process the intervals). This results in running time of  $O((k+1)|E|)$ , where  $k$  denotes the user defined maximum size of a  $pdb$  entry. For the example graph (figure 4) this scheme imposes the processing order:  $a_0, a_2, \dots, a_n, a_1$ . There is still one potential issue with this approach: it does not impose constant bound on queue size. If the lattice graph shows little topological structure (i.e. all vertices other than  $v_0$  and  $v_g$  have exactly  $v_0$  as predecessor and  $v_g$  as successor) the queue can grow up to  $|V| - 2$  vertices in a lattice graph. Redeemingly,  $tv_s$  is trivial in such graphs, as the number of paths is only  $|V| - 2$ .

The  $f$  function compares a prefix  $pre$ 's *target-value* *to-go*  $tv_{suf} = tv - g(pre)$  against the database entry of  $pre$ 's last vertex. Should  $tv_{suf}$  lie inside some interval, there is a chance that an optimal completion of  $pre$  yields precisely the original  $tv$ . Thus, in order to be admissible,  $f$  has to rank such prefixes highest and return 0, else the function returns the distance between  $tv_{suf}$  and the closest bound. As all possible completion lengths lie within the intervals, and all bounds represent actual path lengths, this is the closest any completion of said prefix can come to the original target value. So in essence,  $f$  gives either perfect guidance ( $f(pre) > 0$ ) or no guidance at all ( $f(pre) = 0$ ). Intuitively, if one assumes queries with target values uniformly distributed over the range of the graph's path-lengths, the probability of the former case is inversely proportional to the "area" covered by the intervals. In a DAG, this area increases monotonically in the link-distance from  $v_g$ , as each vertex's database entry covers at least as much ground as each of its successors (figures 5 and 3). Formally:

$$f(pre) = \min_{int \in pdb(pre.last)} (dist(int, tv - g(pre))) \quad (1)$$

Here, an entry is a sorted list of up to  $k$  disjoint intervals, i.e.  $int_i = [lb, ub]$  such that  $\forall i \in 1 \dots k : int_i.lb \leq int_i.ub$  and  $\forall i < j : int_i.ub < int_j.lb$ ,  $dist$  is defined as

$$dist([lb, ub], s) = \begin{cases} 0 & \text{if } lb \leq s \leq ub, \\ \min(|lb - s|, |ub - s|) & \text{else.} \end{cases} \quad (2)$$

In summary, the  $f$  function has the following properties. First, it represents a lower bound on the value of our objective function for the *best* (and thus for all) possible completions of a prefix  $pre$  in  $G$ . Second, for all complete paths  $p \in P_{v_0, v_g}$ ,  $f(p) = dist([0; 0], tv - g(p)) = |tv - g(p)|$  equals the objective function. This allows us to use  $f$  as a prefix evaluation function for heuristic search. Due to the way the pattern database is built,  $f$  is also *monotone*. That is, for any prefix  $pre$  and its descendant  $pre'$ ,  $f(pre) \leq f(pre')$  holds. In particular, for any prefix  $pre$ ,  $f(pre) > 0$  there will be *at least one* immediate successor  $pre'$  with equal value (i.e.

$f(pre) = f(pre')$ ), indicating  $pre'$  is part of the optimal completion of  $pre$ .

## Depth-first Heuristic Target Value Search

In the context of heuristic target-value search with the above  $f$  function, there are two ways in which prefixes can be redundant:

**Duplicity** the optimal completions of any pair of prefixes ending in the *same vertex*, with equal prefix values, will share the same suffix, and have equal deviation from the original target-value. In other words the respective best solutions stemming from said pair will be equal w.r.t. the target-value search's objective function and extra prefixes can therefore be considered redundant and be discarded.

**Domination** For any prefix  $pre$  with  $f(pre) > 0$ ,  $f(pre)$  is the actual deviation of the prefix's best completion from the target value. Therefore, for any pair of prefixes  $pre_a, pre_b$  with  $f(pre_a) > f(pre_b) > 0$ ,  $pre_a$  is dominated by  $pre_b$  and can therefore be discarded.

While it is well known that  $A^*$  is optimal in the number of node expansions for consistent heuristics (such as  $f$ ) (Dechter and Pearl 1985), duplicate detection can be expensive in terms of memory and computational overhead, as in the worst case all previously visited nodes have to be retained. This is especially so for domains like target-value search in which duplicates occur rarely. Here (due to their definition) duplicates are much rarer than in shortest-path searches on the same graph (prefixes ending in the same vertex are not considered duplicates if they have different  $g$ -values).

## The algorithm

To guarantee optimality, we have to generate all prefixes in  $f$ 's "blind-spot" reachable from  $v_0$  (i.e. where  $f = 0$ ). In the worst case, this can be the largest part of the graph's path-space since the number of these prefixes can be exponential in the size of the graph. This is the second culprit that prevents best-first target-value search from scaling to larger graph sizes. At some point the vast share of these prefixes will have to be kept in  $A^*$ 's Open and Closed lists, resulting in a worst-case memory requirement that is exponential in the number of vertices in  $C$ .

To circumvent this, we opted to forego duplicate detection and use an algorithm based on *Depth-First Branch-and-Bound* to search through the blind-spot and determine the prefix  $pre_{opt}$  with lowest  $f \neq 0$  on its "rim". Due to the properties of  $f$ , we know that  $pre_{opt}$ 's optimal completion will be the target-value path. Subsequently we reconstruct the target-value path from  $pre_{opt}$  using a simple *greedy* procedure. We will now describe these algorithms in more detail.

Function DFTVS (listing 1) sets the connection graph (1) and computes a pattern database for  $v_g$  (2). It calls DFBnB, which computes  $pre_{opt}$ ,  $f_{opt}$  and  $tv'_{opt}$  (3).  $pre_{opt}$  and  $tv'_{opt}$  are then fed to EXP to (if necessary) expand  $pre_{opt}$  to the  $tv_p$  (4). Finally, it returns the  $tv_p$  and its deviation from  $tv$  (5).

---

**Algorithm 1:** DFTVS( $v_0, v_g, tv$ )

---

**Input:**  
 $v_0$  : vertex ; // start  
 $v_g$  : vertex ; // goal  
 $tv$  : float ; // target value

**begin**

- 1 | setConnectionGraph( $v_0, v_g$ );
- 2 | buildPdb( $v_g$ );  
| path  $pre := \text{pathFrom}(v_0)$ ;  
| path  $pre_{opt} := \emptyset$ ;  
| float  $f_{opt} := \infty$ ;  
| float  $tv'_{opt} := \infty$ ;
- 3 | DFBnB( $pre, tv, v_g, pre_{opt}, f_{opt}, tv'_{opt}$ );
- 4 | EXP( $pre_{opt}, tv'_{opt}, v_g$ );
- 5 | **return** ( $pre_{opt}, f_{opt}$ );

**end**

---

Note that  $f$ 's signature differs slightly from above: instead of paths, its arguments are a path's last vertex and its target value to-go  $tv'$ , preventing repeated recalculations of path length. Procedure DFBnB first checks whether to end the depth-first traversal (1): that is, if  $pre$  is a path either to goal-vertex  $v_g$  or leads out of the blind-zone ( $f(pre) > 0$ ). If in addition,  $pre$  dominates the current  $pre_{opt}$ , it replaces the latter (2). Otherwise, traversal continues (3): each outgoing edge of  $v$  is concatenated to  $pre$ , the corresponding target-value  $tv'$  computed, followed by a recursive call to DFBnB. If this descent produces a perfect target-value path, the recursion is terminated (5), preventing an unnecessary sweep of  $v$ 's remaining descendants. Finally,  $pre$  is restored to its prior value in preparation for the next edge (6).

Procedure EXP's termination test is whether  $pre$  ends in the goal vertex  $v_g$  (1). Otherwise it computes the best outgoing edge of  $pre$ 's last vertex and its corresponding target value  $tv'$  (2) & (3), concatenates the edge to  $pre$  (4) and recurses on the new  $pre$  (5). The prefix produced by the *depth-first branch and bound search* is either an explicit solution (if it ends in  $v_g$  and thus represents a perfect  $tv$ ), or an implicit solution, whose best completion leads to an optimal solution. In the latter case, a greedy expansion of  $pre$  by its best successor suffices to reconstruct the target-value path.

While the computational worst case complexity remains exponential in the number of vertices in the connection graph  $C$ , the memory requirements are bounded by the number of edges in  $E$  ( $O(|E|)$ ), as in an acyclic graph, the longest  $pre$  can still only contain each edge at most once. This situates DFTVS in *EXPTIME* as opposed to HTVS in *EXPSPACE*.

### Example

Now we will give a small step by step example of a DFTVS query for  $tv = 2.4$ . First, DFTVS sets the connection graph and builds a pattern database as shown in figure 3. Figure 6 shows the call graph for DFBnB and EXP (grey). Each entry comprises the prefix (as  $C$  is not a multi-graph, pre-

---

**Algorithm 2:** DFBnB( $pre, tv', v_g, pre_{opt}, f_{opt}, tv'_{opt}$ )

---

**Input:**  
 $pre$  : path ; //  $v_0 \rightarrow \dots \rightarrow v$   
 $tv'$  : float ; //  $tv - g(pre)$   
 $v_g$  : vertex ; // goal

**Output:**  
 $pre_{opt}$  : path ; //  $v_0 \rightarrow \dots \rightarrow v$   
 $tv'_{opt}$  : float ; //  $tv - g(pre_{opt})$   
 $f_{opt}$  : float ; //  $f(pre_{opt})$

**begin**

vertex  $v := \text{endOf}(pre)$ ;  
float  $val_f := f(v, tv')$ ;

- 1 | **if**  $v = v_g \vee val_f > 0$  **then**
- 2 | | **if**  $val_f < f_{opt}$  **then**  
| | |  $pre_{opt} := pre$ ;  $f_{opt} := val_f$ ;  $tv'_{opt} := tv'$ ;  
| | **return**;
- 3 | **foreach** edge  $e : \text{outEdges}(v)$  **do**  
| | addTo( $pre, e$ );  
| | float  $tv'_{new} := tv' - \text{edgeVal}(e)$   
| | DFBnB( $pre, tv'_{new}, v_g, pre_{opt}, f_{opt}, tv'_{opt}$ );  
| | **if**  $\text{endOf}(pre_{opt}) = v_g \wedge f_{opt} = 0$  **then**  
| | | **return**;
- 6 | | removeLast( $pre$ );

**end**

---

---

**Algorithm 3:** EXP( $pre, tv', v_g$ )

---

**Input:**  
 $tv'$  : float ; //  $tv - g(pre)$   
 $v_g$  : vertex ; // goal

**Output:**  
 $pre$  : path ; //  $v_0 \rightarrow \dots \rightarrow v_g$

**begin**

vertex  $v := \text{lastOf}(pre)$ ;

- 1 | **if**  $v = v_g$  **then**  
| | **return**;
- 2 | float  $val_{min} := \infty$ ;  
| float  $tv'_{best}$ ;  
| edge  $e_{best}$ ;
- 3 | **foreach** edge  $e : \text{outEdges}(v)$  **do**  
| | float  $tv'_{new} := tv' - \text{edgeVal}(e)$ ;  
| | **if**  $f(\text{target}(e), tv'_{new}) < val_{min}$  **then**  
| | |  $val_{min} := f(\text{target}(e), tv'_{new})$ ;  
| | |  $e_{best} := e$ ;  
| | |  $tv'_{best} = tv'_{new}$ ;
- 4 | addTo( $pre, e$ );
- 5 | EXP( $pre, tv'_{best}, v_g$ );

**end**

---

fixes can be represented as stacks of vertices for brevity and clarity,  $v_0$  at the bottom), the related  $f$ -value (in the white ellipsis) and the respective target value to-go (in the grey ellipsis). DFBnB starts its descent with recursive calls I and II. Here the traversal stops immediately, as we have left the

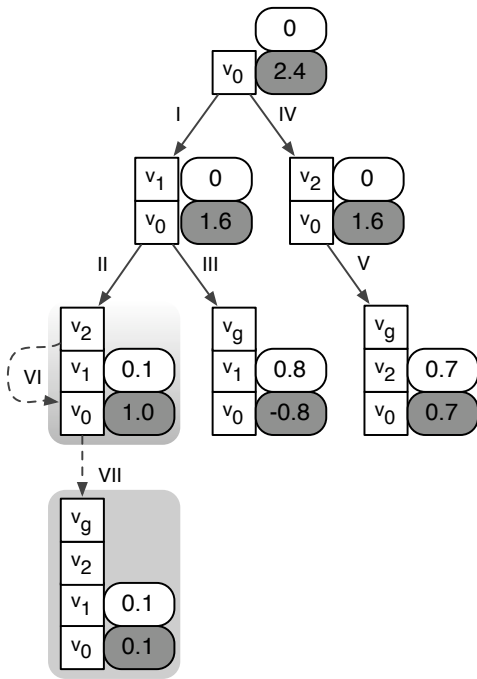


Figure 6: Call graph for a DFTVS query ( $tv = 2.4$ ) on the graph from fig 3.

blind-zone (non-zero  $f$ -value), so DFBnB retains this prefix as intermediate solution  $pre_{opt}$ . The depth-first traversal continues with III, stopping again as expansion has left the blind-zone (prefix ends in  $v_g$  and has  $f \neq 0$ ).  $pre_{opt}$  is not replaced due to its lower  $f$ -value (0.1 vs.0.8). DFBnB’s traversal continues with calls IV and V. Here,  $pre_{opt}$  again is not replaced after comparison of the  $f$ -values.  $pre_{opt}$ , the grey-white node, is the initial calling context of EXP (VI). EXP concatenates the best (and in this case only) successor and calls itself (VII). This final call to EXP will terminate the recursion as  $pre$  ends in  $v_g$ . The stack now holds the  $tpv$  between  $v_0$  and  $v_g$  for  $tv = 2.4$ .

## Evaluation

In the following, we give an empirical evaluation of DFTVS comparing it to BFTVS and inadmissible  $A^*$  with the *shortest path* guiding heuristic on two synthetic test domains: *sparse* and *dense*. All tests were performed on a machine with a 2.8 GHz Intel Core 2 Duo CPU with 4 GB of RAM running MacOS X 10.5.6. We implemented all algorithms as parts of a uniform framework, to allow for fair runtime comparisons.

### Synthetic Domains

Both domains represent connection-graph lattices, consisting of designated start and goal vertices and a “grid” of vertices between them. Generally edge values are assigned randomly (sampled from a uniform  $(0; 1]$  distribution). Both are parameterized in terms of width, height and a seed value for a random-number generator. In the *sparse* domain, vertices

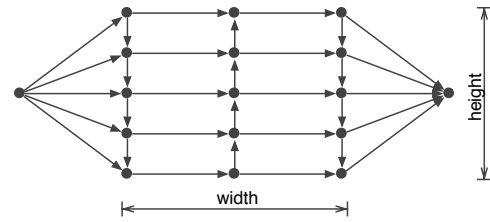


Figure 7: The *sparse* domain: vertices are always connected to their “right”, as well as their “lower” or “upper” neighbours (depending on whether the column is “odd” or “even”).

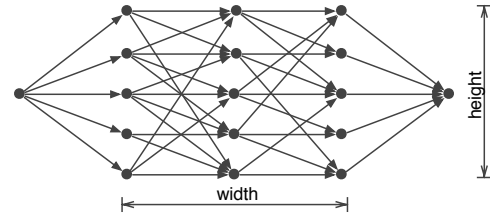


Figure 8: The *dense* domain: vertices are always connected to their “right” neighbour; additionally, for each other vertex in the “right” neighbouring column, there is a connection with probability  $p$ .

(with the exception of  $v_0$  and  $v_g$ ) have a constant out-degree of 2, and path lengths (in number of vertices) between start and goal vary between  $width + 2$  and  $height * width + 2$ . Its general connection pattern is shown in figure 7.

The *dense* domain has uniform path-lengths (in number of vertices) of  $width + 2$ . An additional parameter, probability  $p$ , governs the out-degree of nodes in the grid: a vertex has a connection to a vertex in its “right” neighbour column with probability  $p$  (besides its direct right neighbour, with whom it is always connected). This results in an average out degree of  $p * (width - 1) + 1$ , (which is approximately  $p * \sqrt{|V|}$  for the “square” graphs we mostly use in the evaluation). In general, for “square” graphs, we use the term *dimension* ( $d$ ) to denote *width* and *height* parameters. Also, if not otherwise noted, we allowed up to 5 intervals per *pdb* entry and used 0.5 as *probability* parameter for the *dense* domain.

Both domains are hard in that they contain a large number of paths (exponential in *width* for *dense*, and in  $width * height$  for *sparse*).

Figure 9 shows the average time (in  $\mu\text{sec}$ ), using an  $A^*$  for inadmissible heuristics with  $f(x) = |T - (g(x) + sp(x))|$ , BFTVS (standard  $A^*$  for consistent heuristics with our *pdb*) and DFTVS algorithms with target-values ranging from the shortest to longest path lengths in the underlying *dense* graphs. BFTVS and DFTVS query times include pattern database construction, whereas we did not include the time for computing the shortest-path lengths used by  $A^*$  (they were retrieved from a pre-computed *pdb* at runtime). Each datapoint represents an average over 25 graphs. The runtime distributions reflect the normal distribution of path lengths in the *dense* domain. The problem is hardest if the  $tv$  is

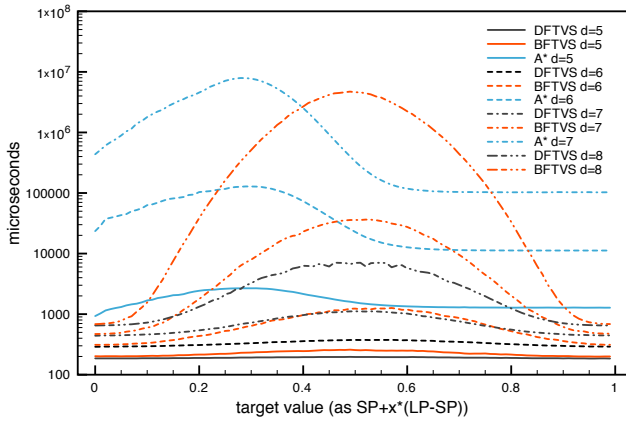


Figure 9: average search times for target values between the shortest and longest path in *dense* graphs of  $A^*$  with the shortest path as guiding heuristic ( $d : 5, 6, 7$ ), BFTVS and DFTVS ( $d : 5, 6, 7, 8$ ).

right between the shortest ( $SP$ ) and longest path ( $LP$ ) of the graph, as most paths come close to the  $tv$  and can only be rejected late by the heuristic, resulting in a large blind-spot. The relative differences in running time widen rapidly from one order of magnitude at  $d = 6$  to three orders of magnitude at  $d = 8$  between BFTVS and DFTVS, so we limited this comparison to very small graphs, with the inadmissible  $A^*$  being between 1 and 3 orders of magnitude worse than BFTVS.

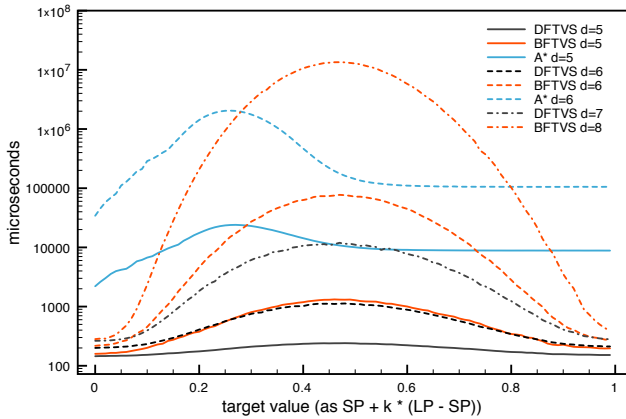


Figure 10: average search times for target values between the shortest and longest path in *sparse* graphs of  $A^*$  with the shortest path as guiding heuristic ( $d : 5, 6$ ), BFTVS and DFTVS ( $d : 5, 6, 7$ ).

Figure 10 gives the average running time for the *sparse* domain, albeit for smaller graphs (up to  $d = 6$  for the  $A^*$  with inadmissible heuristics and  $d = 7$  for BFTVS and DFTVS). Even on these small graphs, search times are about an order of magnitude higher in comparison to *dense* graphs with the same number of vertices.

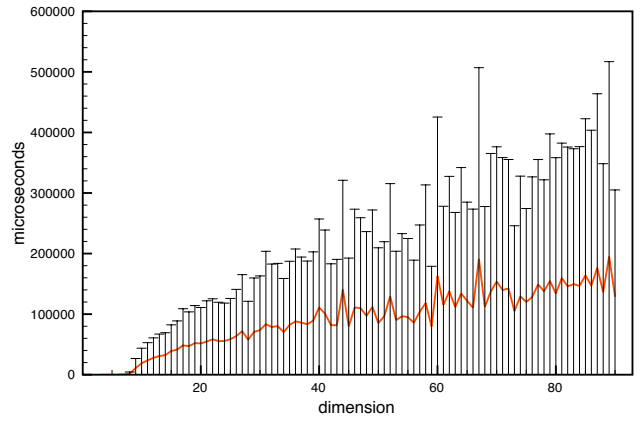


Figure 11:  $DFTVS/dense$ : mean and standard deviation for search times in relation to domain size.

Figure 11 gives an overview of running times for  $DFTVS$  queries ( $\mu =$  red line,  $\sigma =$  error bars) in relation to graph size. For each datapoint we created 10 instances (differing in their seed values) and executed 1000 queries against each, with target-values randomly sampled from a uniform  $[SP; LP]$  distribution. Note how  $DFTVS$ 's  $\mu, \sigma$  for the 8102 vertex  $d = 90$  graph are about 1/10th and half of BFTVS's  $\mu (\sim 9.5 * 10^5), \sigma (\sim 1.3 * 10^6)$  on the 66 vertex  $d = 8$  graph from figure 9.

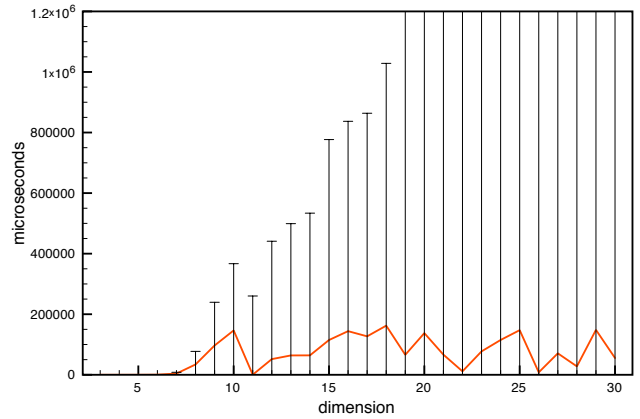


Figure 12:  $DFTVS/sparse$ : mean and standard deviation for search times in relation to domain size.

Figure 12 shows the same for the sparse domain up to 902 vertices graphs ( $d = 30$ ). While the mean is only moderately worse than in *dense*, the standard deviation of query time grows much quicker.

### Pattern Database

Figure 13 gives a comparison of the time (in  $\mu sec$ ) needed to build the pattern database for different graphs (*dimensions* 3-90) between the old method of (Kuhn et al. 2008a) and

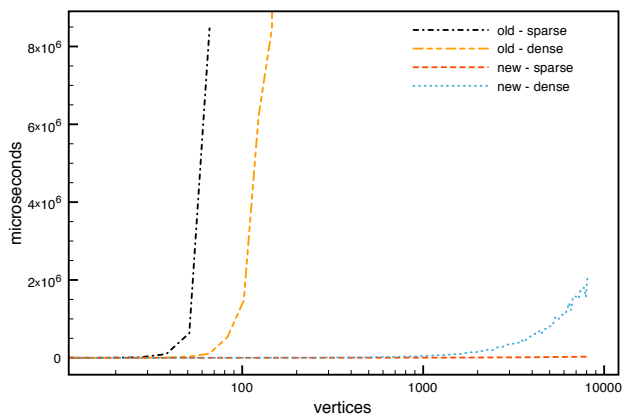


Figure 13: average *pdb* construction times for the old and new approach on the *sparse* and *dense* domains.

our new method. Each datapoint represents an average over 25 graphs (different random seeds). The results show that the high computational overhead limits the old approach to very small graphs, particularly so on the *sparse* domain with its comparatively larger number of paths. The higher cost of the new approach in the *dense* domain is due to the number of edges growing quadratically in the height parameter. Average pattern database construction times were between 50  $\mu$ sec ( $d=3$ ) and 28 msec ( $d=90$ ) for the *sparse* domain.

The larger computation time for building the *pdb* is the only sense in which *dense* is the harder domain. For all other purposes, the much larger number of paths in the *sparse* domain make it a much harder search problem.

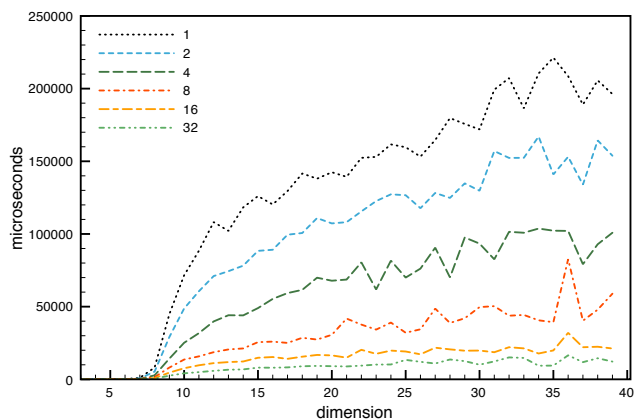


Figure 14: *DFTVS/dense*: means of search times for different maximal *pdb* entry sizes (number of intervals) in relation to domain size.

Figure 14 shows how the number of intervals per *pdb* entry influence search time. With the exception of single and dual intervals per entry, doubling the number of intervals seems to roughly halve the query's runtime. This coincides

with the observations of (Holte and Hernadvolygi 1999) about the relation between size and accuracy of memory-based heuristics.

## Conclusion

In this paper, we have introduced a new approach for target-value path search that allows us to tackle problems at least two orders of magnitude larger than the previous state of the art. We have described an improved method for computing the pattern database in linear time with respect to the number of edges, achieving significant computational savings over our previous approach, which suffers a worst-case exponential time complexity for building the pattern database. The new method also allows trading off memory for increased heuristics fidelity. We have also described a depth-first approach to target-value search that successfully avoids the memory bottleneck in the previous algorithm which relied on best-first search. As demanded by a number of applications, such as on-line diagnosis, the real-time aspect of target-value search can be important. Fortunately, the improved running time and anytime characteristics of the algorithm presented in this paper permit realtime or quasi-realtime solutions to problems of realistic sizes.

## References

- Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.
- Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A\*. *Journal of the ACM (JACM)* 32(3):505–536.
- Dow, P., and Korf, R. 2007. Best-first search for treewidth. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 22, 1146. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.
- Hart, P.; Nilsson, N.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on* 4(2):100–107.
- Holte, R., and Hernadvolygi, I. 1999. A space-time tradeoff for memory-based heuristics. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, 704–709. JOHN WILEY & SONS LTD.
- Kuhn, L.; Price, B.; de Kleer, J.; Do, M.; and Zhou, R. 2008a. Heuristic search for target-value path problem. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*.
- Kuhn, L.; Price, B.; de Kleer, J.; Do, M.; and Zhou, R. 2008b. Pervasive diagnosis: the integration of active diagnosis into production plans. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*.
- Liu, J.; de Kleer, J.; Kuhn, L.; Price, B.; Zhou, R.; and Uckun, S. 2008. A Unified Information Criterion for Evaluating Probe and Test Selection. In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, 1–8.