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. 2008b).

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 amongst others when integrating model-based planning and diagnosis (Kuhn et al. 2008a). where liberties in attaining production goals are exploited 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 edge weights represent the diagnosis engine's confidence that the respective component is working correctly. Assuming single, non-intermittent faults, selecting the path whose predicted success probability is as close as possible to 0.5 maximizes 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 edge values, a *target-value-path* (or tvp in short) 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 closest 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 $P^{tv}_{v_0,v_g} = \underset{|tv-g(p)|}{\operatorname{argmin}} P_{v_0,v_g}$ the set of paths between v_0 and v_g with minimal deviation from tv 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 $P^{tv}_{v_0,v_g}$.

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 union 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 ($O(|V| + |E|)$). 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, for its concatenation with some of its suffixes and the term *optimal completion* of a prefix w.r.t a tv, to denote the completion that is closest to tv.

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 typically leveraged for, e.g., shortest-path problems). While for all possible decompositions $pre \circ suf$ of a tvp w.r.t some tv, suf will be a tvp (from its first vertex to v_g) w.r.t $tv_{suf} = tv - g(pre)$ and pre will be a tvp (from v_0 to the last vertex in pre) w.r.t $tv_{pre} = tv - g(suf)$, tv_{pre} and tv_{suf} are interdependent, and so are the respective *cost functions* for the subproblems. See figure 1 for an example. Worst case, all prefixes in C up to (roughly) value tv will have to be generated during a tvs This leads us to believe that tvs is in $EXPTIME$, as the number of these prefixes

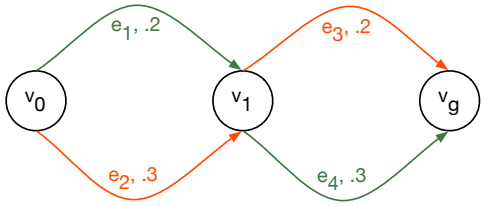

Figure 1: tvs does not exhibit *optimal substructure*: consider the above graph for $tv = 5$. after we expanded v_0 we have two paths $< e_1 >, < e_2 >$ to v_1. Both can lead to optimal solutions with the right completion (i.e. $< e_1, e_4 > and < e_2, e_3 >$), the selection of which depends on the whole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

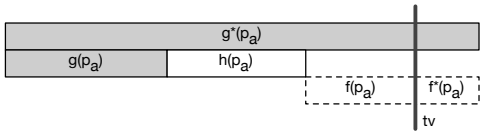

Figure 2: an example for why a heuristic that underestimates suffix lengths will not lead to an admissible tvs heuristic: here $f(p) = |tv - (g(p) + h(p))| > f^*(p) = |tv - g^*(p)|$ while $g(p) + h(p) < g^*(p)$

can be exponential in the number of vertices in C.

Heuristic Target Value Search

A straightforward approach to tackle tvs 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 for tvs due to the *non-linearity* introduced by the absolute value operator. See figure 2 for an example. The basic idea being to find a good solution tvp_{cur} quickly and use it to prune the Open list of all prefixes, whose g value exceeds $tv + |tv - g(tvp_{cur})|$. The search terminates, if either a perfect solution is found (i.e. $g(tvp) = tv$), or the Open list is empty (returning tvp_{cur}).

Previous Work

(Dow and Korf 2007) show, how an admissible heuristic can be constructed for the non-standard objective function of the *treewidth* problem and then be employed in best-first search. (Kuhn et al. 2008b) construct a *pattern database* (Culberson and Schaeffer 1996) 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 used to prune the search tree. Second, given the pattern database for the graph, one can, in addition to guiding the search, detect when the problem of finding an optimal suffix for some prefix degenerates into a shortest or longest path problem, which can then be solved straightforwardly in vertex space. The pattern database (pdb in the following) contains bounds of vertices' different paths' values

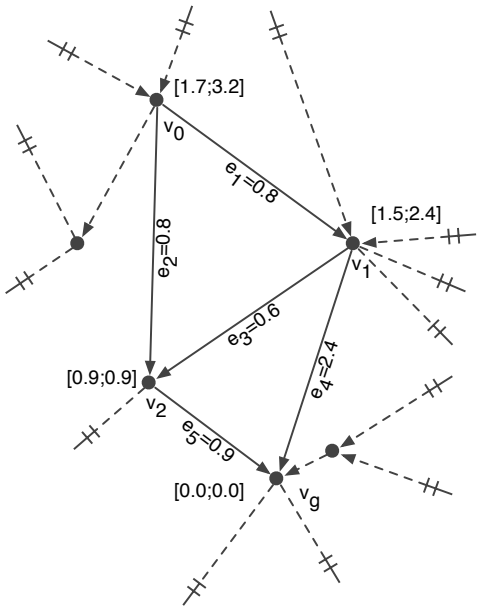

Figure 3: the connection graph (solid edges) of v_0 and v_g with edge values and entries of a single interval *pattern database*

to v_g. Given some prefix pre and tv, one can use the pdb to determine, whether the target value for the suffix $tv - g(pre)$ falls outside the bounds stored in the pdb. If so, the problem of finding an optimal completion for pre breaks down to either a *shortest-* or *longest-path* problem, both of which can be solved using dynamic programming.

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 (more or less) standard A^* with duplicate detection as sketched out above. In contrast to the first approach, which in most cases (if there is no perfect tvp in the graph) has to generate all prefixes in C, with $g < tv + f^*$, this can often make due with a small subset, typically offsetting the cost for constructing the pdb (especially if 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. 2008b) has a worst-case exponential runtime complexity.

In the following, we will show, how to extend the above pdb concept, how to compute such pdb using an algorithm with linear runtime in the size of C and how to apply *depth-first branch and bound search* to tvs. We then give empirical evidence that the combination of these techniques allows us to scale tvs to much larger problems than before.

Pattern Database

The pdb of (Kuhn et al. 2008b) stores a single interval per vertex, with the bounds comprising of 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

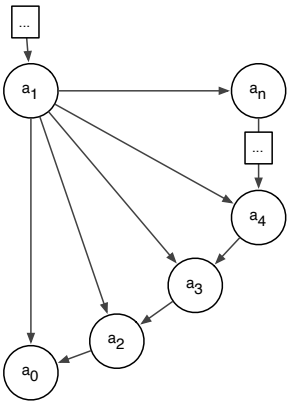

Figure 4: To compute both *shortest* and *longest* paths for its pdb (Kuhn et al. 2008b) proposes a backward propagation scheme that can require exponential time on graphs like this on.

the vertex to v_g). We extend that concept, by allowing the pdb to store multiple, disjoint intervals per vertex, providing a more fine-grained approximation. First, we give a short recapitulation of how the pdb was built in (Kuhn et al. 2008b). The pdb is build using a simple propagation scheme, starting from v_g. Initially, all pdb entries are initialized 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 in-edge e, the lower bound of the predecessor is set to the min of its former lower bound and the sum of $value(e)$ and the lower bound of v. The same is done for the upper bound (using max). If the predecessor's pdb entry changes in the process, it is added to the queue. The algorithm terminates, once the queue is empty.

This is one of the two major culprits in preventing the scaling of tvs to large graphs (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 this order). Now a_1 is expanded, and all its predecessors (not shown in the figure) are placed on the queue. Next is a_2. Here as of our assumptions, a_1 gets updated and is thus placed on the queue, along with a_3. Now all the predecessor of a_1 are processed, until we encounter, again, a_1 on the queue, resulting in its updated predecessors being added to the queue 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 runningtime for constructing the pattern-database.

This can be avoided by using a dynamic programming approach, as each pdb 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 of the pdb ensures that all successors entries are already available. During construction, we store a counter with each vertex, initialized with its number of successors in the connection graph C (technically we only need to create/initialise the counter, once we first encounter the vertex 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: First, all successor's intervals are transformed, by adding the value of the respective edge to all bounds. Second, any overlap between the transformed intervals of all successors is resolved by computing the smallest covering intervals. Third, while number of intervals exceeds a user-defined maximum, the two closest intervals are fused. See figure 5 for an example.

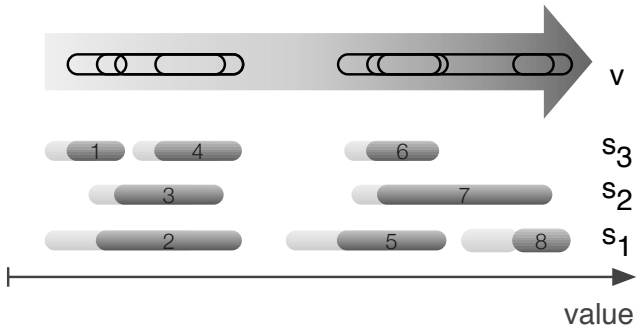

Figure 5: pdb entry construction for some vertex v with successors s_1, s_2, s_3: successor ranges are shifted by the value of the resp. connecting edge. These intervals are accessed in ascending order of their lower bounds (denoted by the numbers). We begin by fetching the first interval and set it 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 C 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 especially true for its successors, so v's successor counter will be 0, 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 follows, that each edge will be accessed twice (in the predecessor direction to update the counts and the other way 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: $a_0, a_2, \ldots, a_n, a_1$. There is still one potential issue with this approach: it does not impose any constant bounds on queue size. If the lattice graph shows little topological structure (i.e. all vertices other then 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, tvs 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* $tv' = tv - g(pre)$ against the pdb entry of pre's last vertex. Should tv' 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' 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 either gives perfect guidance ($f(pre) > 0$) or no guidance at all ($f(pre) = 0$). Intuitively, assuming tvs 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 pdb entry covers at least as much ground as each of its successors. (see figure 5 and, for an example figure 3). Formally:

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

Here, a pdb entry is a sorted list of up to k intervals $int_i = [lb, ub]$ 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

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

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 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 prefix evaluation function for heuristic search. Due to the way the pdb is built, f is also *consistent*. 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 tvs 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 $g(pre)$s (for these prefixes) 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 with regards to tvs's objective function and (either) one of the 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 pre's best completion from the original tv. 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 in which duplicates occur rarely, such as tvs, where (due to their definition) duplicates are much rarer compared to 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, i.e. the number of these prefixes can be exponential in the size of the graph. This is the second culprit that prevents best-first tvs 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 Search* 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 tvp. Subsequently we reconstruct the tvp from pre_{opt} using a simple *greedy* procedure. We will now describe these algorithms in more detail:

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 := pathFrom(v_0);
path pre_{opt} := $\emptyset$;
float f_{opt} := ∞;
float tv'_{opt} := ∞;
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

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 tvp (4). Finally, it returns the tvp and its deviation from tv (5).

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

```
Input:
pre : path ;                          //  v_0 → ... → v
tv' : float ;                         //  tv − g(pre)
v_g : vertex ;                        //  goal
Output:
pre_opt : path ;                      //  v_0 → ... → v
tv'_opt : float ;                     //  tv − g(pre_opt)
f_opt : float ;                       //  f(pre_opt)
begin
    vertex v := endOf(pre);
    float val_f := f(v,tv');
1   if v = v_g ∨ 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 : outEdges(v) do
        addTo(pre, e);
        float tv'_new := tv' - edgeVal(e)
4       DFBnB(pre, tv'_new, v_g, pre_opt, f_opt, tv'_opt);
5       if endOf(pre_opt) = v_g ∧ f_opt = 0 then
            return;
6       removeLast(pre);
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 its depths-first traversal (1): that is, if pre is a path either to v_g or leads out of the blind-zone ($f(pre) > 0$). If in addition, pre dominates pre_{opt}, the latter is displaced (2). Otherwise, the traversal continues (3): Iteratively, each outgoing edge of v is concatenated to pre, the corresponding tv' computed, followed by a recursive call to DFBnB. If this descent produced a perfect tvp, 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 v_g (1). Otherwise it computes the best outgoing edge of $pre's$ last vertex and its corresponding tv' (2) & (3), concatenates the edge pre (4) and calls itself 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 tvp), 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 build the tvp.

While the computational worst case complexity remains exponential in the number of vertices in C, the memory requirements are bounded by the number of edges in E

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

```
Input:
tv' : float ;                         //  tv − g(pre)
v_g : vertex ;                        //  goal
Output:
pre : path ;                          //  v_0 → ... → v_g
begin
    vertex v := lastOf(pre);
1   if v = v_g then
        return;
    float val_min:=∞;
    float tv'_best;
    edge e_best;
2   foreach edge e : outEdges(v) do
        float tv'_new := tv' - edgeVal(e);
3       if f(target(e), tv'_new) < val_min then
            val_min := f(target(e), tv'_new);
            e_best := e;
            tv'_best = tv'_new;
4   addTo(pre, e);
5   EXP(pre, tv'_best, v_g);
end
```

($O(|E|)$), as in an acyclic graph, the longest pre can still only contain each edge at most once. This situates DFTVS in $EXPTIME$.

Example

Now we will give a small step through example of a DFTVS query for $tv = 2.4$. First, DFTVS sets the connection graph and builds an pdb as shown in figure 3. Figure 6 shows the call graph for DFBnB and EXP (grey). Each entry comprises of the prefix (as C is not a multi-graph, prefixes 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 is first stopped as we have left the 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 displaced 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 displaced 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 tvp 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* as guiding heuristic on two synthetic test domains

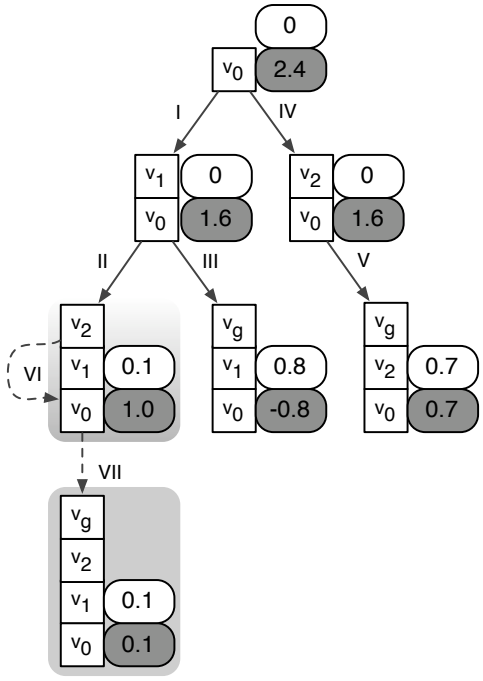

Figure 6: Call graph for a DFTVS query ($tv = 2.4$) on the graph from fig 3.

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 Mac OS X 10.5.6. We implemented all algorithms as parts of a uniform framework, to allow for fair runtime comparisons.

Synthetic Domains

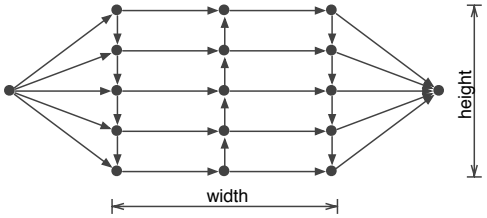

Figure 7: The *sparse* domain: vertices are always connected to their "right", as well as their "lower" or "upper" neighbors (depending on whether the column is "odd" or "even"

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 (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

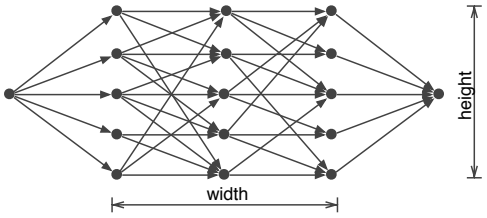

Figure 8: The *dense* domain: vertices are always connected to their "right" neighbor; additionally, for each other vertex in the "right" neighboring column, there is a connection with probability p

$height * width + 2$. 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" neighbor column with probability p (besides its direct right neighbor, 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*).

Search

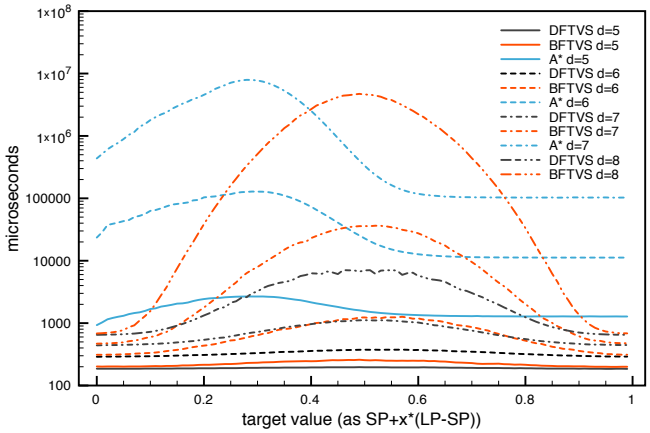

Figure 9: average search times for target values between the shortest and longest path in *dense* graphs of A^* with the shortest path as guiding heuristic ($d : 5, 6, 7$), BFTVS and DFTVS ($d : 5, 6, 7, 8$).

Figure 9 shows the average time (in μsec), 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 queries include pdb 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 data point 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 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 then BFTVS.

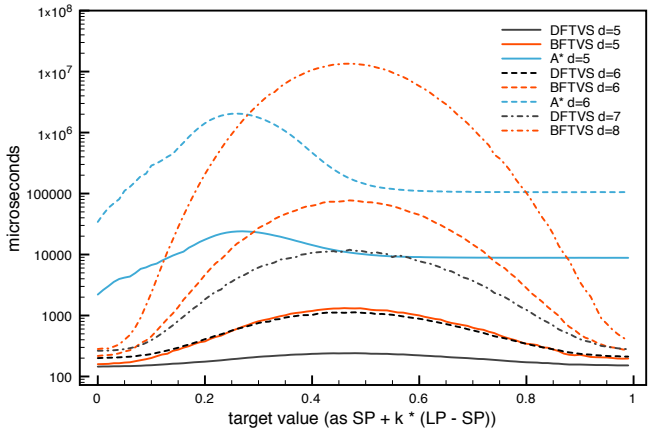

Figure 10: average search times for target values between the shortest and longest path in *sparse* graphs of A^* with the shortest path as guiding heuristic ($d : 5, 6$), BFTVS and DFTVS ($d : 5, 6, 7$).

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

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

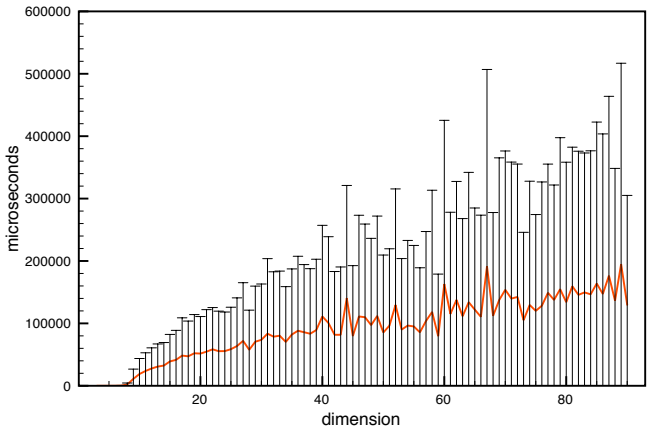

Figure 11: $DFTVS/dense$: mean and standard deviation for search times in relation to domain size.

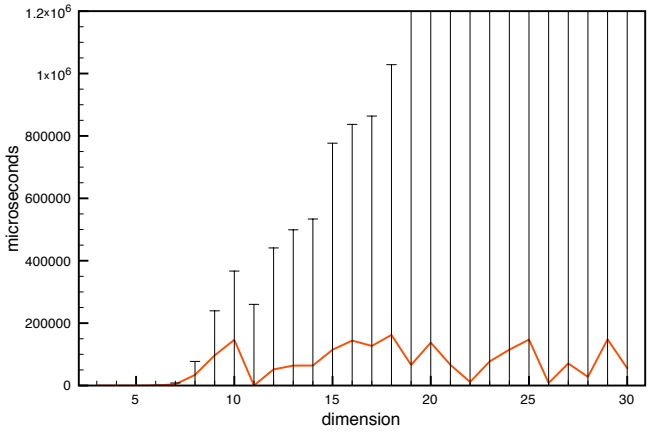

Figure 12: $DFTVS/sparse$: mean and standard deviation for search times in relation to domain size.

grows much quicker.

Pattern Database

Figure 13 gives a comparison of the time (in μsec) needed to build the pattern database for different graphs ($dimensions$ 3-90) using the old method of (Kuhn et al. 2008b) and our new method. Each data point 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 amount of edges in the $dense$ domain growing quadratically in the height parameter. Average construction times were between 50 μsec (d=3) and 28 $msec$ (d=90) for the $sparse$ domain.

The larger pdb computation time for building the pdb is the only sense in which $dense$ is the harder domain. For all other purposes, the much larger of paths in the sparse

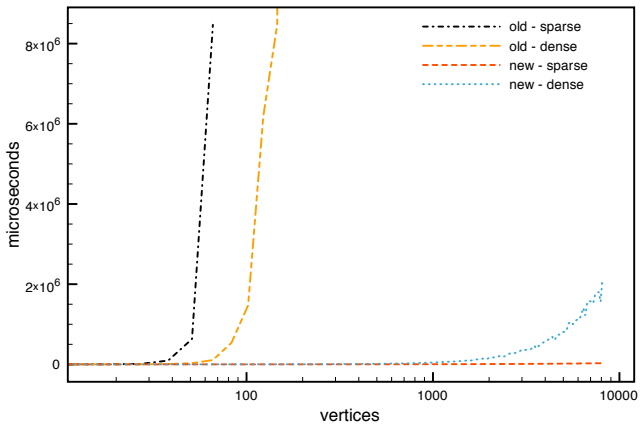

Figure 13: average *pdb* construction times for the old and new approach on the $sparse$ and $dense$ domains.

domain make it a much harder search problem.

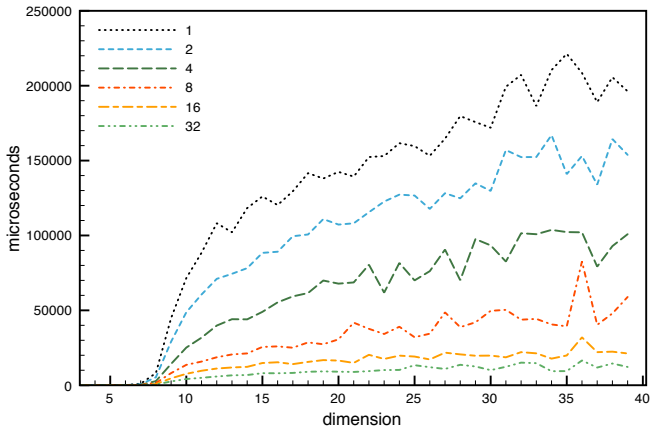

Figure 14: $DFTVS/dense$: means of search times for different maximal *pdb* entry sizes (number of intervals) in relation to domain size.

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 half the query's runtime. This coincides with the observations of (Holte and Hernádvölgyi 1999) about 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 the previous approach, which suffers a worst-case exponential time complexity for building the pattern database. We have also described a depth-first approach to target-value search that successfully avoids the memory bottleneck in the previous algorithm that uses 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 contributions of this paper make it possible to solve target-value search problems of realistic sizes in realtime or quasi-realtime.

References

Culberson, J., and Schaeffer, J. 1996. Searching with pattern databases. *Lecture Notes in Computer Science* 1081:402–416.

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 Hernádvölgyi, I. 1999. A space-time tradeoff for memory-based heuristics. In *Proceedings of the National Conference on Artificial Inteligence*, 704–709. John Wiley & Sons Ltd.

Kuhn, L.; Price, B.; de Kleer, J.; Do, M.; and Zhou, R. 2008a. Pervasive diagnosis: the integration of active diagnosis into production plans. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*.

Kuhn, L.; Price, B.; de Kleer, J.; Schmidt, T.; Zhou, R.; and Do, M. 2008b. Heuristic search for target-value path problem. In *The First International Symposium on Search Techniques in Artificial Intelligence and Robotics*.

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.