Target-Value Search Revisited

Carlos Linares López
Planning and Learning Group
Universidad Carlos III de Madrid
28911 Leganés (Madrid) - Spain
carlos.linares@uc3m.es

Roni Stern
SEAS
Harvard University
Cambridge, MA 02138 USA
roni.stern@gmail.com

Ariel Felner
Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel 85104
felner@bgu.ac.il

Abstract

This paper addresses the *Target-Value Search* (TVS) problem, which is the problem of finding a path between two nodes in a graph whose cost is as close as possible to a given target value, T. This problem has been previously addressed only for directed acyclic graphs. In this work we develop the theory required to solve this problem optimally for any type of graphs. We modify traditional heuristic search algorithms for this setting, and propose a novel bidirectional search algorithm that is specifically suited for TVS. The benefits of this bidirectional search algorithm are discussed both theoretically and experimentally on several domains.

1 Introduction

Given a target value T, the *Target-Value Search* problem (TVS) consists of finding a path in the search space from a start state, s to the goal state, t, such that the cost of the path is as close as possible to a given value, T. The problem was recently introduced [Kuhn *et al.*, 2008; Schmidt *et al.*, 2009], motivated by several applications, such as when planning a tour of a given duration in a park. Another application for TVS is diagnosis systems, where it has been shown that finding the most informative test can be reduced to finding a test of whose probability of success is equal to 0.5 [Liu *et al.*, 2008].

Previous work has addressed TVS only in the context of directed acyclic graphs (DAGs) [Kuhn *et al.*, 2008]. In addition, they performed only small-scale experiments, with graphs having at most 70 nodes.

In this work, we address TVS for general graphs. We discuss the unique properties of TVS in comparison with a shortest path (SP) search problem. In particular, we describe how the role of an admissible heuristic in TVS differs from its role in SP. Then, we describe how to modify existing SP search algorithms such as A* and IDA* to TVS, and propose a novel bidirectional approach especially suited for TVS. Lastly, we describe how to define and use abstraction-based heuristics for TVS, introducing the notion of *Induced Transition Graph*. Experimental results on a number of classical search benchmarks show the benefit of the proposed bidirectional search coupled with the *Induced Transition Graph* heuristic over the other approaches.

2 Problem Definition

In the *target value search* problem (TVS) we are given a graph $G = (V, E)$, two states $s \in V$ and $t \in V$ and a *target value* $T \in \mathbb{N}$. The task is to find a path P in G from s to t such that the cost of the path $c(P)$ is as close as possible to T. Given a path P we denote its length with $|P|$. Also, we define $\Delta_T(P) = |T - c(P)|$. Thus, a solution to TVS is a path P that minimizes $\Delta_T(P)$. We denote this minimal Δ_T by Δ_T^*, and refer to a solution with $\Delta_T(P) = 0$ as a *perfect solution*.

Since previous work on TVS addressed DAGs, only simple paths, i.e., paths containing every vertex at most once, were considered as possible solutions. By contrast, paths in general graphs can contain the same vertex or edge many times. Thus, there are several ways to define TVS for general graphs:

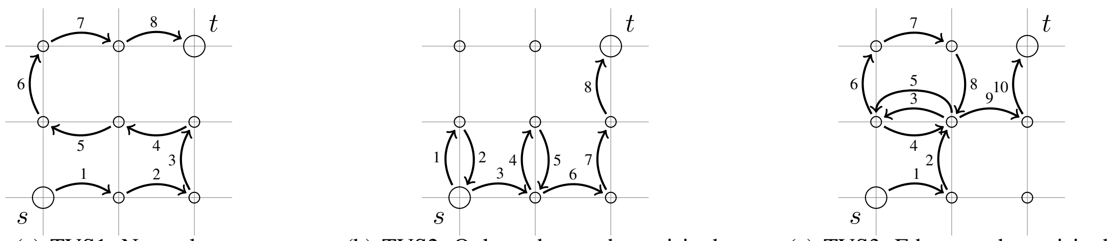
- **TVS1:** Only simple paths are allowed.
- **TVS2:** Revisiting vertices is allowed, but every edge can only be visited once.
- **TVS3:** Every path is allowed —both states and edges can be revisited.

Figure 1 shows one solution for each of TVS1, TVS2 and TVS3 for the TVS problem where G is a 3×3 grid, s is the bottom left corner, t is the upper right corner and $T = 9$. Note that there is no perfect solution, so that solutions with length 8 or 10 are equally preferable.

Choosing the most suited TVS variant depends on the exact problem and domain that is being addressed. In this work we address TVS2 (nodes can be revisited but not edges), and focus on undirected graphs with unit edge cost.

2.1 Previous Approach for TVS

As mentioned above, previous work has addressed TVS only in the context of DAGs [Kuhn *et al.*, 2008]. For every node n, they first stored the lowest and highest cost path from s to t that passes via n. These values were stored in a data structure similar to a pattern database (PDB) [Culberson and Schaeffer, 1996]. This PDB was then used in an A*-like search to prune nodes that can not be part of paths that have a smaller Δ_T than the current best solution. Later work generalized this



(a) TVS1: No cycles (b) TVS2: Only nodes can be revisited (c) TVS3: Edges can be revisited

Figure 1: Solutions to the different TVS variants in a 3×3 grid with $T = 9$

PDB to hold several intervals of possible path costs for every node in the PDB.

This previous approach can not be used directly for TVS2 on general graphs for several reasons. Remarkably, creating the PDB requires traversing every possible path between s and t. In DAGs, this can be done efficiently with dynamic programming but is exponentially harder for general graphs, which may contain cycles, as well as for very large implicitly given graphs [Schmidt *et al.*, 2009].

3 A* and IDA* for TVS

In this section, we modify two classical search algorithms, namely A* [Hart *et al.*, 1968] and IDA* [Korf, 1985]. It is important to remark that, in the context of TVS, none of these algorithms perform any kind of *duplicate detection* since different paths to the same node can have different suffixes or continuations, depending on the nodes previously visited by every path.

3.1 TVSA*

A* is a best-first search algorithm that uses as an evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the sum of the edge costs from s to n and $h(n)$ is a lower bound on the path cost from n to t. A* maintains two lists of nodes, an open list (OPEN) and a closed list (CLOSED). Initially, OPEN contains s and CLOSED is empty. In every iteration of A*, the node with the lowest f in OPEN is moved to CLOSED and its children are generated. A generated child is added to OPEN, unless it was already in OPEN or CLOSED with a lower g-value. This pruning of previously generated nodes is called *duplicate detection*. A* halts when t is expanded, in which case $g(t)$ is the lowest cost from s to t [Hart *et al.*, 1968].

Next, we describe our adaptation of A* to TVS, which we call TVSA*. Pseudo-code for TVSA* is given in Algorithm 1. Since we aim at finding the path with the lowest Δ_T, the nodes in TVSA* are expanded according to a different evaluation function than in A*. Let $\Delta_T(n) = |g(n) - T|$. TVSA* expands the node from OPEN that has the lowest $\Delta_T(n)$. However, it first considers *only* the nodes in OPEN with $g(n) \leq T$ (line 5). If no such nodes exists, TVSA* chooses the node with the lowest $\Delta_T(n)$ among the nodes with $g(n) > T$ in OPEN.

Whenever a goal node is found, $best_\triangle$ is updated with its Δ_T if it is smaller than the current $best_\triangle$ (line 13). Nodes with $f(n) \geq T + best_\triangle$ can never lead to a solution with lower Δ_T than $best_\triangle$, and are thus pruned (line 11). The search halts when either a perfect solution is found or a node with $g(n) \geq T + best_\triangle$ is chosen from OPEN (line 9).

Algorithm 1: TVSA*

```
   Input: $s$, the start state
   Input: $t$, the goal state
   Input: $T$, the target value
   Output: A path $P$ from $s$ to $t$ with minimal $\Delta_T(P)$
1  OPEN$\leftarrow \{s\}$
2  $best_\triangle \leftarrow \infty$
3  while OPEN is not empty do
4      if OPEN contains a node with $g(n) \leq T$ then
5          $best \leftarrow \arg\min_{n \in OPEN, g(n) \leq T} \{\Delta_T(n)\}$
6      else
7          $best \leftarrow \arg\min_{n \in OPEN} \{\Delta_T(n)\}$
8      if $g(best) \geq T + best_\triangle$ then
9          return $best_\triangle$
10     if $f(best) \geq T + best_\triangle$ then
11         Continue
12     if $(best = t)$ then
13         $best_\triangle \leftarrow min(best_\triangle, |T - g(best)|)$
14         if $best_\triangle$=0 then return $best_\triangle$
15         ;
16     Expand($best$)
17 return $best_\triangle$
```

Theorem 1 *TVSA* is optimal and complete.*

Proof. When TVSA* halts either (1) a perfect solution is found (line 14), or (2) OPEN is empty (line 17) or (3) a node with $g(n) \geq T + best_\triangle$ is chosen for expansion (line 9). If a perfect solution is found then clearly no better solution exists. If OPEN is empty, then all paths from s to t must have been enumerated, and thus the best Δ_T found is the smallest possible. If a node with $g(n) \geq T + best_\triangle$ is chosen for expansion, this means that all nodes with $g < T + best_\triangle$ were expanded, then future solutions found by the search can only have a cost that is larger than $T + best_\triangle$. Thus, the Δ_T of these solutions would be larger than $best_\triangle$. □

3.2 TVSIDA*

Iterative Deepening A* (IDA*) is a known SP algorithm [Korf, 1985] which mimics A* by performing a sequence of depth-first searches (DFS). Each DFS is limited by a given threshold, such that nodes with f-value larger than that threshold are not expanded. If a goal is expanded, the search halts. Otherwise, a new DFS starts with a larger threshold, set to be the lowest f-value seen that was above the previous threshold.

Next, we describe our modification of IDA*, which we call TVSIDA*. When solving shortest path problems, A* has an advantage over IDA* due to duplicate detection. In TVS this advantages disappears, making an adaptation of IDA* to TVS more appealing, potentially enjoying IDA*'s benefits over A*, such as linear memory requirements and faster time per node. TVSIDA* behaves as follows. First, a DFS iteration is executed with a threshold set to be exactly T. If a solution is found (of length less or equal to T), its Δ_T is stored in a variable called $best_\triangle$. If $best_\triangle = 0$, a perfect solution was found and the search can halt. Otherwise, subsequent iterations of DFS are performed, incrementing the threshold just like in IDA*, i.e., updating the current threshold to be the minimum f-value seen above the current threshold. Whenever a solution is found, its Δ_T is computed and $best_\triangle$ is updated if the Δ_T of the new solution is lower than the previous $best_\triangle$. TVSIDA* halts when reaching a threshold that is greater than or equal to $T + best_\triangle$. Correctness proof follows the proof given above for Theorem 1 and is omitted due to lack of space.

4 Bidirectional TVS (BTVS)

Consider the differences between A* and TVSA*. A* searches the *state space*, while TVSA* searches a space that we call the *path space*, where every node corresponds to a path from s. Naturally, the state space is substantially smaller than the path space. Thus, A* will find a path to t faster than TVSA*. However, A* may not find the path to t with the smallest Δ_T while TVSA* is guaranteed to find this path according to Theorem 1. Following, we propose a unique bidirectional search that attempts to enjoy the complementary benefits of state space and path space searches. In particular, this algorithm reaches t fast like a *state space search*, and is guaranteed to return a solution with the lowest Δ_T like a *path space search*. We call this algorithm Bidirectional Target Value Search (BTVS).

BTVS is composed of two alternating searches: a *forward search* and a *backward search*. The forward search starts from s and is performed on the state space, while the backward search starts from t and is performed on the path space. Importantly, the backward search, which potentially searches a substantially larger space than the forward search, is limited to only consider paths that are composed of nodes and edges found by previous forward searches. BTVS alternates between the forward and the backward search until Δ^*_T is found. Next, we describe BTVS in details.

4.1 Forward Search

The forward search is a regular uniform-cost search (*aka* Dijkstra's search) from s to t. Duplicate detection is performed, and for every visited node n we store the lowest g-value found from s to n. This g value found for n in the forward search is denoted by $g_f(n)$. When t is expanded, the lowest-cost path to t was found and $g_f(t)$ is its cost. The Δ_T of this solution is stored in $best_\triangle$. Next, the backward search is called.

Algorithm 2: High-level BTVS

```
1 Forward search until the lowest cost path is found
2 Backward search checking all paths in the induced graph
3 Forward search until all nodes with $f < T + best_\triangle$ have been
  expanded
4 Backward search checking all paths in the induced graph
5 return $best_\triangle$
```

4.2 Backwards Search

The backwards search consists of running TVSIDA* from t to s but considering only expanded nodes in the forward search (i.e., the nodes in CLOSED). We call the graph that is composed of these nodes and the edges between them the *induced graph* of the forward search. The backward search is done in the path space (i.e., no duplicate detection is done), possibly considering paths that were pruned by the forward search. During the backward search $best_\triangle$ is updated if a solution is found that has a lower Δ_T than the current $best_\triangle$. The backward search halts only after all paths in the induced graph of the forward search were visited or a perfect solution was found in which case BTVS can halt.

In addition, the following pruning rule is performed during the backwards search. It is based on the observation that $g_f(n)$ is the cost of the shortest path from s to n. Let $g_b(n)$ be the cost of the backward path from t to n. If $g_f(n) + g_b(n) \geq T + best_\triangle$ then the backward search prunes n, since continuing the backward search from n to s will only find paths at least as large as $g_f(n)$, and thus a path with a better Δ_T than $best_\triangle$ can not be found.

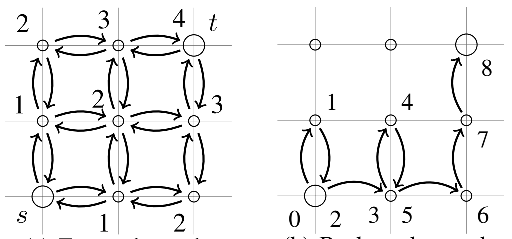
To speed up the backwards search, direct children of a node n (which were direct anscestors of n in the forward search) are sorted in increasing order of their deviation from the target value T, computed for every child node c by $g_f(c) + g_b(c)$. This node ordering heuristic is intended to give preference to nodes more likely to lead to paths with low Δ_T. We observed empirically that this enhancement is beneficial.

4.3 The BTVS Algorithm

Algorithm 2 describes in a high-level manner how BTVS uses the forward and backward searches described above. First, the forward search finds a lowest cost path from s to t (line 1). Then, the backward search tries to find a path with lower Δ_T, considering only the induced graph of the forward search (line 2). Then, the forward search continues, expanding all the nodes with $g \leq T + best_\triangle$ (line 3). Finally, a second backward search is done to check all paths passing through these nodes (line 4). We use the term *BTVS iteration* to denote a single call for the forward search and then the backward search. Only two iterations are performed and after them, the path with the lowest Δ_T is guaranteed to be found.

Theorem 2 BTVS *is optimal and complete.*

Proof. If the first iteration does not find a perfect solution, then the second forward search visits all the nodes with $g < T + best_\triangle$. These are all the nodes visited by TVSA*. Thus, the backward search will check all the possible paths from s to t that would be checked by TVSA*, and thus it is



(a) Forward search (b) Backwards search

Figure 2: The first iteration of BTVS for solving a pathfinding problem from $(0, 0)$ to $(2, 2)$ with $T = 9$ in a 3×3 grid

guaranteed to find the same quality of path as TVSA*. Thus, following Theorem 1 BTVS is also optimal and complete. □

Figure 2 shows the first iteration of the BTVS algorithm on the same problem shown in Figure 1. Figure 2(a) shows the induced graph of the forward search, where the g_f-cost from s to reach every node is recorded. The forward search ends with finding the shortest path P^* between s and t with a cost equal to 4 units, so that the current deviation is $\Delta_T(P^*) = |9-4| = 5$. In the backward search, DFS with a threshold equal to the target value, 9, is started but only considering nodes in CLOSED. The lowest Δ_9 found by the backward search has a cost of 8 so that the new $best_\Delta = |9-8| = 1$. Since a perfect solution has not been found, the forward search of the second iteration expands all nodes with $g < T + best_\Delta = 9+1 = 10$.

There can be many variations of BTVS, which alternate differently between the forward and backward searches. The variation described above is both optimal and complete, and has performed better than other simple variants we have experimented with.

5 Abstractions and the Induced Transition Graphs

Experimental results reported in the next section show that BTVS is worse than TVSIDA*. We identified two potential sources of inefficiency that may explain this behavior. First, consider the backward search phase of BTVS. It is applied after the forward search, and is intended to find solutions with better Δ_T than $best_\Delta$. Naturally, it might be the case that no better solution exists in the induced graph of the forward search, and thus the backward search was redundant. Applying the backward search redundantly is one source of potential inefficiency of BTVS.

Another source of potential inefficiency is the forward search in the second iteration of BTVS (line 3 in Algorithm 2). In BTVS, this forward search continues until all nodes with $g < T + best_\Delta$ are expanded. However, when the forward search expands a node, it may add more edges and nodes to the induced graph of the forward search. Thus, it is possible that applying the backward search even before all the nodes with $g < T + best_\Delta$ are expanded is beneficial.

Both inefficiencies could be remedied if one had an oracle that would know when running the backward search may find a better solution (i.e., one with $\Delta_T < best_\Delta$). Next, we describe a novel heuristic method that suggests when the backward search might lead to an improved solution. We prove

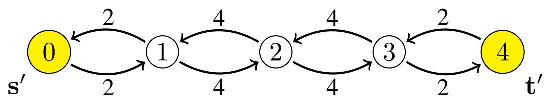


Figure 3: Induced Transition Graph of the forward search shown in Figure 2(a)

that this method can detect with certainty when the backward search can not lead to an improved solution, thus saving redundant calls to the backward search. This method is based on the *Induced Transition Graph* (ITG), defined next. We denote the combination of BTVS and ITG by T*.

5.1 Induced Transition Graph

An ITG is an abstraction of a state space $\mathcal{S}$ that is defined with respect to the information gathered during the forward search. In particular, in an ITG all the nodes with the same g_f value are mapped into a single node. We label every node in the ITG according to the g_f value of the nodes in the original search space that are mapped to it. An edge between nodes i and j in the ITG exists iff the forward search encountered an edge between a node n with $g_f(n) = i$ to a node m with $g_f(m) = j$. Every edge (i, j) in the ITG is labeled with the number of times in the forward search that a node with $g_f(\cdot) = i$ generated a descendant with $g_f(\cdot) = j$. We denote the label of edge (i, j) as $\#(i, j)$. Let s' and t' be the nodes in the ITG s and t are mapped to. Clearly, $g_f(s') = 0$. It is easy to see that the ITG can be built during the forward search with a constant additional overhead.

Figure 3 shows the ITG of the CLOSED list generated by the forward search shown in Figure 2(a) where the nodes s' and t' are highlighted.

Next, we describe how the ITG can be used to identify cases where the backwards search of BTVS is not needed and the incumbent solution, i.e., the path P seen so far with the lowest $\Delta_T(P)$, can be safely returned as the optimal solution to TVS. We denote the incumbent solution by P_{best}, and note that $\Delta_T(P_{best}) = best_\Delta$.

A path P' in the ITG is called an ITG traversal if it is a path from s' to t' such that every edge (i, j) exists in P' at most $\#(i, j)$ times. It is easy to see that every path from s to t that is found by the backwards search in the original state space has a corresponding ITG traversal of the same length.

Thus, instead of performing the backwards search to check if it contains a better path than P_{best}, one can enumerate all the ITG traversals and check whether there is an ITG traveral P' such that $|T - |P'|| < best_\Delta$.

Theorem 3 *If for every ITG traversal P' it holds that $|T - |P'|| \geq \Delta_T(P_{best})$ then the backwards search will not provide any solution better than P_{best} and it can be avoided.*

Proof. Proof by contradition. Assume that the above condition holds, and that there exists a path P'' that will be found by the backwards search having $\Delta_T(P'') < \Delta_T(P_{best})$. This means that there exists an ITG traversal P' that corresponds to P'' with $|T - |P'|| = |T - |P''|| < \Delta_T(P_{best})$, contradicting the condition that $|T - |P'|| \geq \Delta_T(P_{best})$. □

Theorem 3 can be used to restrict the forward and backward searches in the second iteration and thus, to address the inefficiencies identified in BTVS as follows. Firstly, the forward search of the second iteration invokes the ITG every time that all nodes with $f(n) = g_f(s,t) + k$ have been expanded with $k \in \mathbb{N}$. In case the ITG returns a deviation δ such that $f(n) \geq T + \delta$ the forward search can immediately terminate since the value returned by the ITG is a lower bound on the deviation wrt T. In this case, the backwards search of the current iteration can be issued returning the optimal solution to the TVS. Another possible outcome is that the ITG returns a null deviation meaning that there is potentially an optimal solution. In this case, the backwards search is run also but if the optimal solution is not found, T* enters the next iteration resuming the forward search in exactly the same point it left it —i. e., expanding the next node in OPEN. Finally, the TVSIDA* systematically issued by BTVS in the last iteration can be safely skipped if every ITG traversal *GT* has a deviation which is larger or equal than the deviation of the incumbent solution.

Note that the ITG concept can be applied to any state space abstraction that preserves edges. We demonstrate it on g^*-based abstractions as a domain-independent solution.

5.2 Using ITGs in Unit Edge Cost Domains

In this section we assume unit edge costs and show how to check efficiently if there is an ITG traversal with $\Delta_T < best_\Delta$. This is done by using *Dynamic Programming* and exploiting a specific structure of the ITG occuring only in any unit edge cost domain.

In unit edge cost domains, the ITG consists of a chain of nodes as shown in Figure 3. This is because if all the edges have unit cost, then two adjacent nodes in the original graph can only have g_f values that differ by one or zero. Therefore, the length of the longest path between t' and $s' = 0$ in the ITG is:

- Let $l_{i-1,i}$ denote the number of transitions between $(i-1)$ and i in the ITG and $l_{i,i-1}$ the number of transitions in the opposite direction, e. g., $l_{0,1} = l_{1,0} = 2$ in Figure 3.
- Let c_i denote the number of self-loops observed from i to itself.

Let L_i denote the length of the longest path to reach 0 from node i in the ITG without taking self-loops into account and using only nodes from 0 to i. The length of such path can be easily computed as:

$$L_i = L_{i-1} + \left\{ \begin{array}{ll} 2l_{i,i-1} - 1 & \mathit{if}\, l_{i-1,i} \geq l_{i,i-1} \\ 2l_{i-1,i} + 1 & \mathit{otherwise} \end{array} \right.$$

where $L_0 = 0$. Then, the length of the longest path between t' and 0 in the abstract state space using only nodes in the range $[0, t']$ is $L_{t'} + C_{t'}$ where $C_{t'} = \left(\sum_{j=1}^{t'} c_j\right)$, if self-loops are taken into account.

However, the preceding result does not account for the longest path between the goal node t' and 0 since it only considers nodes in the range $[0, t']$. The following reasoning results from the observation that if T is the target value from t' then $(T - \delta)$ is the target-value after following a path of length δ. Let g_{max} denote the leftmost node in the ITG from which a transition has been observed, e.g., $t' = 4$ in Figure 3. Obviously, if $L_{g_{max}} + C_{g_{max}} < T - (g_{max} - t')$ then $T - (g_{max} - t') - (L_{g_{max}} + C_{g_{max}})$ is a lower bound on the deviation wrt T since the longest path from g_{max} is shorter than the target value at g_{max}. Otherwise, if $L_{g_{max}} + C_{g_{max}} \geq T - (g_{max} - t')$, then the parity rule is applied: if g_{max} has the same parity than $T - (g_{max} - t')$ then the lower bound is necessarily 0; otherwise, it is assumed to be 1 unless there are self-loops (as they break parity) and the lower bound is 0.

6 Experimental Results

We evaluated the performance of TVSA*, TVSIDA* and T* (BTVS with ITG) on three standard search benchmarks: 4-connected grid pathfinding, the tile puzzle and the pancake puzzle. We also report the number of solved instances by BTVS to prove the usefulness of the ITGs . In every experiment, a start state and goal state were chosen such that the shortest distance between them is typically two. This was done to emphasize the complexity of the TVS problem, as finding a solution with a specific value can be very challenging even for states that are close to one another. All the experiments have been performed on a Linux computer with a time cutoff of 120 seconds and 2 Gb of memory.

6.1 4-Connected Grids Pathfinding

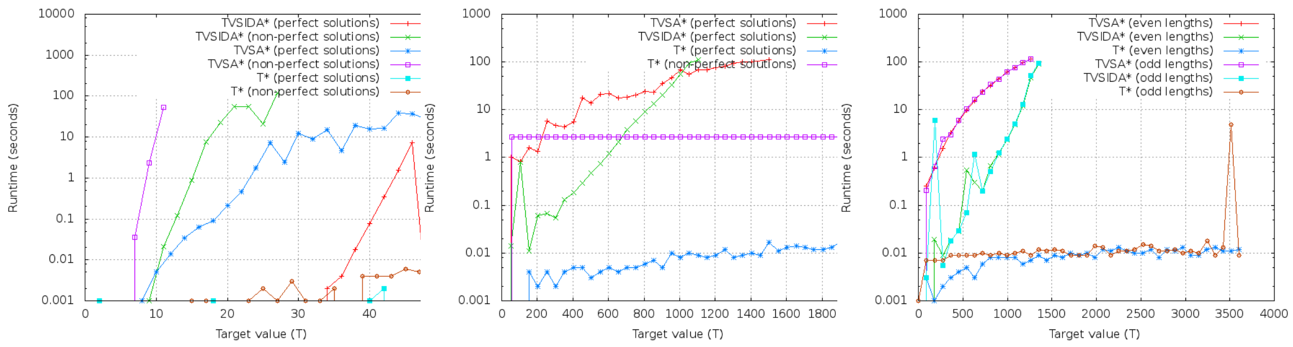
We first experimented with pathfinding in a 4-connected grid. We used random 512x512 grids from the benchmark suite of [Sturtevant, 2012], with 10%, 20%, 30% and 40% blocked cells. Ten instances were randomly generated in every grid. All target values in the range $[2, 50]$ were tried. The Manhattan distance heuristic was used for TVSA* and TVSIDA*.

Table 1 shows the number of instances solved by each algorithm, under our time and memory limitations. We differentiated between instances where a perfect solution existed (even T) and where such a solution did not exist, odd T. As expected, those instances where there is no perfect solution were harder for all the algorithms. This is because when a perfect solution is found, all algorithms can immediately halt. By contrast, when no perfect solution exists, all algorithms need to check all paths with cost $T + best_\Delta$ to verify that the path with the lowest Δ_T was found. The results from Table 1 show clearly that T* is more efficient than the other algorithms, solving substantially more instances. In general, TVSA* performs worse than TVSIDA*.

There is no remarkable difference in relative performance for varying degrees of obstacles. Therefore, only the average running time over 10 random instances per target value in a map with a 30% of obstacles is shown in Figure 4(a). When plotting this Figure, the average is computed only over the solved instances and this explains the decaying profile exhibited by TVSA* and TVSIDA* for high values of T. For other ratios of filled squares the same trend is verified: while TVSIDA* and TVSA* solve less cases as T increases, T* solves them all always in less than 0.006 seconds each, i. e., 3 and 4 orders of magnitude faster respectively.

6.2 The 8-puzzle

Ten instances were selected by hand to ensure that symmetries were avoided: eight of them at an optimal distance of



(a) Runtime, grid w. 30% of obstacles (b) Runtime, 8-puzzle (c) Runtime, 6-pancake

Figure 4: Average running time of all algorithms over 10 problems with different target values

	Perfect				Not-Perfect			
Blocked	0.1	0.2	0.3	0.4	0.1	0.2	0.3	0.4
TVSA*	213	180	199	201	40	43	45	50
TVSIDA*	250	250	248	239	81	88	97	123
BTVS	250	250	250	250	80	86	96	130
T*	250	250	250	250	240	240	240	240

Table 1: # Instances solved, 512x512 grid

four moves and 2 of them of two moves. In total, 80 different target values in steps of 50 were tried, 40 starting from $T = 4$ (having perfect solutions) and 40 starting with $T = 5$ (no perfect solutions). As a heuristic for TVSA* and TVSIDA* we used the perfect heuristic function built with a PDB that mapped all tiles except the blank. However, they solved only 35.8% (286), and 28.8% (230) of the problems, respectively. While BTVS solved only 30 problems, T* solved all of the 800 instances. In this domain, T* runs five orders of magnitude faster than the other algorithms as shown in Figure 4(b).

Note that while TVSA* and TVSIDA* use the perfect heuristic, T* does not use any heuristic. However, TVSA* and TVSIDA* could solve only ten instances when no perfect solution is available, and T* solved all these cases in less than 2.7 seconds on average.

6.3 The Pancake Puzzle

We experimented on pancake puzzles with 5, 6, 7, 8 and 9 pancakes. Ten random instances were generated for each puzzle at distance two from the target. 40 even target values from $T = 2$ and 40 odd ones starting from $T = 3$ were used in all cases uniformly distributed until $T = (N-1)N!$, which is the diameter of an N-pancake puzzle. This totaled in 4,000 experiments.

TVSA* and TVSIDA* used the *gap* heuristic [Helmert, 2010] which is known to be very accurate for these sizes of pancake puzzles. They solved 26.57% (1,063) and 24.17% (967) of the cases respectively, while T* solved 99.45% (3,978) of them. Table 2 shows the total number of solved problems by all algorithms.

For all sizes of pancake puzzles, T* was able to solve much more problems than TVSA* and TVSIDA*. Moreover, any problem that were solved by either TVSA* or TVSIDA* was also solved by T* and much faster. In the larger pancake puzzles (with 8 and 9 pancakes), TVSA* and TVSIDA* failed completely to solve problems with target values larger than $T = 2$ or $T = 3$. Thus, the 6-Pancake has been chosen to show the general trend on the overall running time – see Figure 4(c). In this case, T* ran four and five orders of magnitude faster than TVSIDA* and TVSA*, respectively

	even				
	5	6	7	8	9
TVSA*	346	143	20	10	10
TVSIDA*	309	140	45	10	10
BTVS	42	10	10	10	10
T*	400	400	400	400	400
	odd				
	5	6	7	8	9
TVSA*	353	141	20	10	10
TVSIDA*	268	119	46	10	10
BTVS	40	10	10	10	10
T*	378	400	400	400	400

Table 2: # instances solved, pancake puzzle

T* failed in 22 cases in the 5-Pancake for large odd values of T near the maximum value. This effect was not observed for larger pancakes since the difference between the target value increased when distributing them uniformly. In fact, it was observed that, for all algorithms, the overall effort to find solutions is different if T is even or odd though there are always optimal solutions.

7 Conclusions

In this work we addressed the TVS problem for general graphs. We described two algorithms for it that are based on the well-known A* and IDA* algorithms. In addition, we proposed a novel bidirectional algorithm for TVS called BTVS which runs a forward search to find nodes and edges in the graph, and a backward search to find paths with cost closer to the target value. An abstraction-based method called ITG is then introduced to decide intelligently when to switch to the backward search. The combination of BTVS and ITG, named T*, shows exceptional results on three well-known domains.

References

[Culberson and Schaeffer, 1996] Joseph C. Culberson and Jonathan Schaeffer. *Advances in Artificial Intelligence*, chapter Searching with pattern databases, pages 402–416. Springer-Verlag, 1996.

[Hart *et al.*, 1968] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100–107, 1968.

[Helmert, 2010] Malte Helmert. Landmark heuristics for the pancake problem. In *Symposium on Combinatorial Search (SOCS-10)*, pages 109–110, Atlanta, Georgia, United States, July 2010.

[Korf, 1985] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Kuhn *et al.*, 2008] Lukas Kuhn, Tim Schmidt, Bob Price, Johan de Kleer, Rong Zhou, and Minh Do. Heuristic search for target-value path problem. In *The First International Symposium on Search Techniques in Artificial Intelligence and Robotics*, 2008.

[Liu *et al.*, 2008] J. Liu, J. de Kleer, L. Kuhn, B. Price, R. Zhou, and S. Uckun. A unified information criterion for evaluating probe and test selection. In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pages 1–8. IEEE, 2008.

[Schmidt *et al.*, 2009] Tim Schmidt, Lukas Kuhn, Bob Price, Johan de Kleer, and Rong Zhou. A depth-first approach to target-value search. In *Symposium on Combinatorial Search (SOCS-09)*, 2009.

[Sturtevant, 2012] Nathan Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.