Heuristic-Driven "Treasure Hunt" with Linear Performance Guarantees[*]

Yury V. Smirnov, **Sven Koenig**, and **Manuela M. Veloso**
Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213

{smir,skoenig,veloso}@cs.cmu.edu

Abstract

Consider the task of reaching a goal state in a partially or completely unknown domain. To accomplish such a task search algorithms have to explore the domain sufficiently to locate a goal state and a path leading to it, performing therefore what we call uninformed "treasure hunt." Very often prior knowledge in the form of heuristic values estimating distances towards the goal state is readily available. A heuristic-driven strategy can be very successful: it can significantly outperform uninformed exploration-oriented approaches. However, heuristic values can be misleading: if we model the domain as a graph, the worst-case complexity of a heuristic-driven algorithm can grow faster than linear on the weight of the graph (sum of all edge lengths).

Known exploration approaches can solve the uninformed "treasure hunt" problem with linear performance guarantees, but they cannot utilize heuristic values. Furthermore, their average-case (empirical) performance is usually worse than that of heuristic-driven approaches even for uninformed problems. In its turn, the complexity of known heuristic-driven algorithms can be worse than linear, if heuristics are misleading.

We develop a new algorithmic framework for the heuristic-driven "treasure hunt," called VECA, that combines the advantages of both approaches. We show that VECA provides linear performance guarantees, and that these guarantees do not deteriorate the average-case performance.

[*]This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and the National Science Foundation under grant number IRI-9502548. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations or the U.S. government.

1 Introduction In this paper we investigate (1) how to learn unknown graphs efficiently, and (2) how to utilize provided heuristic values to guide goal-directed exploration. First problem corresponds to an on-line version of the Chinese Postman problem (OnCPP):

> Consider the task of traversing all edges of an unknown strongly connected weighted directed graph at least once and then returning to the starting vertex. One always has a map of all vertices that one has visited so far, can recognize the if one sees then again, and knows how many unexplored edges leave each visited vertex (but does not know which vertices they lead to until one explores them too). How can one explore such graph efficiently?

The second problem corresponds to what is known in literature as the "treasure hunt." It differs from OnCPP in that a goal state (goal states) is established, and one is supposed to reach the goal. It can be also viewed as a search problem with uncertancies.

These graph exploring problems are simple and natural optimization problems that are often encountered in practical problems, for example, robot navigation. In particular, it occurs in the following situations: (A) An autonomous robot is put into an unknown office building and has to learn a topological map of its new corridor environment; (B) A software agent has to find a World Wide Web page of a given content by following links from its current page.

If one abstracts from low level sensing and control issues, one can reformulate the robot exploration problem as an OnCPP, thus, making this problem amenable to theoretical graph-learning approaches. Since corridors can be traversed in both directions, the resulting graph is undirected. In some problems after traversing a directed edge, an agent does not learn how to traverse an opposite edge. This happens, for example, in secured buildings with two-side doors having different opening codes, or in learning of a new-born when a success in learning how to perform an action (switch off a TV) does not immediately imply a skill of undoing the same action (turn on a TV). This fact encourages us to investigate also the exploration of bi-directed domains.

2 Problem Descriptions We use the following notations for both exploration and search problems: $G = (V, E)$ is a graph with finite number of vertices (or states), $v_{start} \in V$ is the start state, and $GOAL \subseteq V$ is the non-empty set of goal states. $A(v) \subseteq E$ is the set of directed edges that can be traversed from $v \in V$. Edge $e \in A(v)$ has positive length $length(e) > 0$, its traversal results in successor state $succ(e)$. The goal distance $gd(v)$ of v is the smallest length of a path in G following which a goal state can be reached from v. We assume that the goal distance of every vertex in G is finite. We further assume that graph G is bi-directed, meaning that each directed edge has an opposite edge (called the "twin" of the edge). The weight of the graph is $weight = \sum_{v \in V} \sum_{e \in A(v)} length(e)$, the sum of the lengths of all directed edges.

If $e \in A(v)$ is unexplored, then $length(e)$ and $succ(e)$ are unknown. To explore an edge, the algorithm has to traverse it. We assume that this reveals only $length(e)$ and $succ(e)$, but no additional information. Initially, heuristic knowledge about the effects of actions is available in form of estimates of the goal distances. Classical AI search algorithms attach heuristic values to states. This would force us to evaluate an unexplored edge $e \in A(v)$ according to the heuristic value of s, since both $length(e)$ and $succ(e)$ are not yet known. We therefore attach heuristic values $h(e)$ to edges instead; they are estimates of $length(e) + gd(succ(e))$, the shortest path from v to a goal state when first traversing e. If all $h(e)$ are zero, then the algorithm is uninformed.

The problem of learning an unknown graph and the goal-directed exploration problem can now be stated as follows:

> **The Problem of Learning an Unknown Graph**: Explore all edges of graph G and return to the starting vertex $v_{start} \in V$.
>
> **The Goal-Directed Exploration Problem**: Get an agent from s_{start} to a state in G if all actions are initially unexplored in all states, but heuristic estimates $h(s, a)$ are given.

In both problems, we use the length of the exploration walk to measure the complexity of an algorithm and are interested in how this length depends on the weight of the graph (the sum of all edge lengths). This is a realistic measure, because the cost of moving often significantly prevails the cost of deliberation.

3 Previous Approaches Various exploration approaches has been discussed in the literature. They can be divided into two main groups: uninformed exploration approaches and heuristic-driven search approaches.

3.1 Uninformed Exploration Approaches Uninformed exploration approaches explore all edges. They have no notion of goal states and do not use heuristic values to guide the exploration process. However, they can be used for “treasure hunt”, because they visit all vertices of the graph. One can just stop the exploration when one hits a goal state. The following algorithm has been used earlier as part of proofs on Eulerian tours, for example by Hierholzer[2]. Recently it was re-considered by several researchers as a graph learning algorithm [1, 4]:

> **Building a Eulerian Tour Algorithm (BETA):** Traverse unexplored edges whenever possible (ties can be broken arbitrarily). If all edges emanating from the current vertex have been explored, execute the initial sequence of edge traversals again, this time stopping at all vertices that have unexplored emanating edges and apply the algorithm recursively from each such state.

Note, that any exploration algorithm that operates on a Eulerian domain and traverses unexplored edges whenever possible, can be *stuck* (find no emanating unexplored edges available at a current vertex) for the first time only at the starting vertex. Therefore, retracing its initial traversals is possible for such exploration.

The same argument is true for all recursive procedures. BETA is similar to Depth-First Search in some sense, but – instead of backtracking its latest moves when it gets stuck – it repeats the initial walk, because backtracking is not always possible on arbitrary Eulerian graphs. BETA executes every action at most twice (see, for example [1]) which implies the following theorem:

Theorem 1 *BETA explores any Eulerian unknown graph with the complexity of at most two weights of the graph.*

Since BETA can be applied to the "treasure hunt" problem too, it solves the problem with the complexity that is linear on the weight of the graph. The first example of an unknown graph in Figure 1 shows that two weights of the graph is the tight bound for BETA: it can traverse a pair of directed edges with length w, at which point it gets stuck at its starting vertex. BETA has then to re-traverse both edges with length w again in order to explore edges of length 1 and come back to the starting vertex. The complexity of BETA is $4w + 2$ and the weight of the graph is $2w + 2$. The ratio of the two quantities approaches 2 for large w. Thus, the complexity of BETA does not improve, when one considers bi-directed graphs instead of arbitrary Eulerian graphs. The second example in Figure 1 shows that the complexity of BETA also asymptotically approaches two weights of the graph in the "treasure hunt" problem.

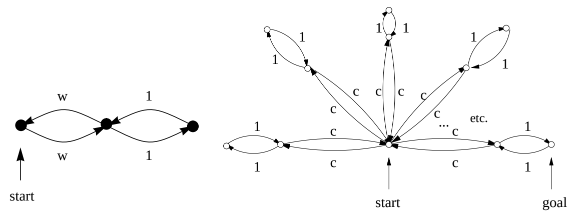

Figure 1: Two Worst-Case Examples for BETA

BETA cannot utilize heuristic values to guide the search process towards a goal state, although heuristics can cut down search time. However, BETA provides a gold standard for evaluating "treasure hunt" algorithms, since no uninformed algorithm can do better in the worst case.

3.2 Heuristic-Driven Algorithms AI researchers have long realized that heuristic knowledge can be a powerful tool to cut down search effort. It can be utilized effectively to guide the search process towards a goal state. One of the most efficient heuristic-driven algorithms is based on the combination of the nearest neighbor principle and one the most appealing search methods for classical Search-in-Memory problems A*[7]. Various versions and approximations of this algorithm has been repeatedly re-discovered under different names. This represents the idea behind such algorithms as Incremental Best-First Search[8] and the Dynamic A* algorithm[11]. The Learning Real-Time A*[5] and Prioritized Sweeping[6] are fast approximations of similar algorithms.

> **Agent-Centered A* Algorithm:** Consider all paths from the current vertex to an unexplored edge e. Among these paths, select one with minimal cost, where the cost of a path is defined as the sum of the path's length plus the heuristic value $h(e)$ (break ties arbitrarily). Traverse the path from the current state to an unknown edge, explore the edge, and repeat the process until a goal state is reached.

AC-A* is very versatile: It can be used to search completely known, partially known, or completely unknown domains, it is able to perform goal-directed exploration efficiently in "dynamic" environments that experience occasional changes. Algorithms using the nearest neighbor (greedy) heuristic have good empirical complexity, they often outperform algorithms with better worst-case complexity in average over a series of runs both for the regular off-line Travelling Salesman problem [9] and for the on-line Chinese Postman Problem [3]. However, the worst-case complexity of uninformed and informed AC-A* is at least $\Omega(\log |V| / \log \log |V|)$ times the weight of graph $G = (V, E)$, while BETA guarantees $O(weight(G))$[3, 10].

3.3 Our Results We develop an algorithmic framework, the parameterized Variable Edge Cost Algorithm (VECA), that solves both exploration problems, can accommodate a variety of exploration and search strategies on undirected or bi-directed domains, but is able to guarantee that every pair of opposite edges is traversed only a certain number of times. VECA can be implemented easily, make the same optimal performance guarantees as BETA for the lowest possible parameter, and allows one to trade-off smoothly more freedom of the exploration or search strategy against a smaller worst-case complexity. Empirical evidence suggests that VECA does not deteriorate average-case complexity of exploration or heuristic-driven "treasure hunt", but to the opposite often outperforms other strategies in sparely connected domains, and when heuristic values are misleading.

4 Variable Edge Cost Algorithm We have developed a framework for goal-directed exploration, called the Variable Edge Cost Algorithm (VECA), that can accommodate a wide variety of uninformed and heuristic-driven exploitation algorithms (including AC-A*). VECA relies on the exploitation algorithm and thus on the heuristic values until they prove to be misleading. It monitors the behavior of the exploitation algorithm and uses a parameter k to determine when the freedom of the exploitation algorithm should get restricted. If an action and its twin together have been executed k times or more, VECA restricts the choices of the exploitation algorithm on that part of the state space, thus forcing it to explore the state space more. As a result, VECA switches gradually from exploitation to exploration and relies less and less on misleading heuristic values.

The same framework can be applied to OnCPP without significant changes. One has just to choose an appropriate exploitation algorithm for step 3 or 3', for example uninformed AC-A*. In this case the algorithm stops exploring and returns

```
Input:
a goal-directed exploration problem,
the value of VECA's parameter $k$ (a non-negative, even integer),
and a heuristic-driven exploitation algorithm (to be used in step 3).

Basic-VECA uses three variables for each edge $e = (v, w)$: $count(e)$ keeps track of how often the action has
been traversed, $reserve(e)$ is its reserved VECA cost, and $cost(e)$ is its actual VECA cost.
  1. Set $count(e) := cost(e) := 0$ for all $e \in E$. Set $i := 0$ and $v := v_{start}$.
  2. If $v \in GOAL$, then stop successfully.
  3. Consider all cycle-less paths starting at $s$ and ending with an *unexplored edge*. Select one with minimal
     VECA cost from these paths, using the heuristic-driven exploitation algorithm to break ties.
  4. Consider all actions in the action sequence, one after another. For each action $a$ in the sequence, do:
     (a) Traverse $e$. Let $e' = (w, v) \in E$ be the twin of $e = (v, w)$.
     (b) Set $count(e) := count(e) + 1$.
     (c) If $count(e) + count(e') = 1$, then set $i := i + 1$ and afterwards $reserve(e) :=$
         $reserve(e') := 2^{-i}$.
     (d) If $count(e) + count(e') \geq k$ and $cost(e) = 0$, then set $cost(e) := cost(e') :=$
         $reserve(e)$.
     (e) If $count(e) + count(e') > k$, then set $cost(e) := \infty$.
     (f) Set $v := succ(e)$.[a]
  5. Go to step 2.
                    Alternatively, step 3 can be replaced by:
  3' Consider all cycle-less paths starting at $s$ and ending with an edge whose VECA cost is zero. Select one
     with minimal VECA cost from these paths, using the heuristic-driven exploitation algorithm to break
     ties.
```

[a]According to our notations $succ(e) = w$.

Figure 2: The Basic-VECA Framework

to the starting vertex after all edges have been explored, as indicated by the absence of unexplored edges emanating from visited vertices. Since it appears that the same framework is applicable to both problems, the rest of the discussion is devoted exclusively to the more intriguing problem of the heuristic-driven "treasure hunt".

We describe VECA in two stages. We first discuss a simple version of VECA, called Basic-VECA, that assumes that traversing an edge also identifies its twin. Later, we drop this assumption. Basic-VECA is described in Figure 2. It maintains a cost for each edge that is different from its length. These VECA costs guide the search. Initially, all of them are zero. Whenever Basic-VECA traverses a pair of twin edges for the first time (i.e. it explores one of the two edges for the first time and has not yet traversed the other one), it reserves a VECA cost for them, that will later become their VECA cost. The first pair of twin edges gets a cost of 1/2 reserved, the second pair a cost of 1/4, the third pair a cost of 1/8, and so on. Basic-VECA assigns the reserved cost to both twin edges when it traverses the pair for the kth time (or, if $k = 0$, when it explores an edge of the pair for the first time). Whenever the pair is traversed again, Basic-VECA assigns the just traversed edge (but not its twin) an infinite VECA cost, which effectively removes it. The VECA costs are used as follows: Basic-VECA always chooses a sequence of edges with minimal VECA cost that leads from the current vertex to an unexplored edge (step 3) or, alternatively, to an edge with zero VECA cost (step 3'). The exploitation algorithm is used to break ties. Initially, all VECA costs are zero and there are

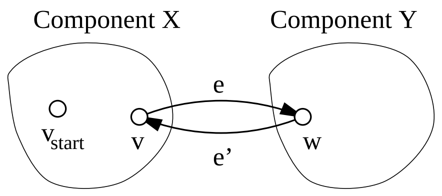

Figure 3: A Simple Example of Graph Exploration

lots of ties to break. The more edges Basic-VECA assigns positive VECA costs to, the fewer ties there are and the less freedom the exploitation algorithm has.

To gain an intuitive understanding of the behavior of Basic-VECA, consider a simple case, namely a tree-shaped state space, and assume that Basic-VECA uses step 3. Figure 3 shows a pair of twin edges, e and e', that connect two components of the tree, X and Y. The exploitation algorithm can traverse $e = (v, w)$ freely until it and its twin $e' = (w, v)$ together have been traversed a total of k times. Then, Basic-VECA assigns both edges the same positive VECA cost. At this point in time, the algorithm is located in X (the component that contains the start state), since k is even and the algorithm alternates between both components. If Y does not contain any more unexplored edges, neither e nor e' will be traversed again. Otherwise there is a point in time when Basic-VECA traverses e again to reach one of those unexplored edges. When this happens, Basic-VECA prevents the exploitation algorithm from leaving Y until all edges in Y have been explored (this restriction of the freedom of the exploitation algorithm constitutes a switch from exploitation to exploration): Because Y can only be entered by traversing e, this edge was explored before any edge in Y. Consequently, its reserved VECA cost, which is also its current VECA cost, is larger than the reserved VECA cost of any edge in Y. The reserved VECA costs are exponentially decreasing: $2^{-i} > \sum_{j>i} 2^{-j}$ for all finite $i, j \geq 0$. Thus, any path that does not leave Y has a smaller VECA cost than any path that does, and Basic-VECA cannot leave Y until all of Y's edges have been explored. When Basic-VECA has finally left Y, the VECA costs of e and e' are infinite, but there is no need to enter or exit Y again.

In general, Basic-VECA traverses every pair of twin edges a total of at most $k + 2$ times before it finds a goal state. This implies the following theorem:

Theorem 2 *Under the assumption that an edge traversal identifies the twin of the edge, Basic-VECA, with even parameter $k \geq 0$, solves any goal-directed exploration problem with a cost of $O(1) \times weight$ (to be precise: with a cost of at most $(k/2 + 1) \times weight$).*

Proof[Sketch]: Both Minimal Spanning Tree (MinST) and Maximal Spanning Tree (MaxST) of the explored portion of the graph play important roles in VECA.

First, MinST of the explored portion of the graph provides the set of shortest pairwise distances with the precision up to zero ties (zero-cost paths). In other words: If pairwise shortest distances form a cycle $x \rightarrow y \rightarrow z \rightarrow x$, this cycle should be of zero VECA cost, where $x \longrightarrow y$ denotes a path from vertex x to vertex y. We prove this by contradiction: Suppose that one of the cycle paths contains an edge with non-zero cost, then pick an edge with the biggest VECA cost from the cycle $x \longrightarrow y \longrightarrow z \longrightarrow x$, let it be in path $x \longrightarrow y$. Then path $x \longrightarrow z \longrightarrow y$ becomes strictly cheaper than ($x \longrightarrow y$)-path and the algorithm cannot consider $x \longrightarrow y$ as part of the shortest path for any pair of vertices. Thus, the set of shortest paths in the explored portion of the graph forms a tree. However, we cannot state anything about the relations of VECA costs for MinST-edges.

On the other hand, MaxST corresponds to edge traversals that lead to the first visit of vertices according to VECA. For MaxST we know that paths from the root (starting vertex) to other vertices have strictly monotone (decreasing) sequence of VECA costs.

Let $ZC(v)$ be a zero-component of vertex v – a maximal set of vertices reachable from vertex v by paths of zero VECA cost. Basic-VECA possesses the following properties:

1. Pairs of edges traversed $k+1$ or $k+2$ times establish a stack behavior, i.e. if $e_1 = (v_1, w_1)$ and $e'_1 = (w_1, v_1)$ had been traversed for the $k+1$st time before $e_2 = (v_2, w_2)$ and $e'_2 = (w_2, v_2)$ were traversed for the $k+1$st time, then the later pair e_2 and e'_2 should be traversed for the $k+2$nd time (in the direction opposite to the $k+1$st traversal) before the $k+2$nd traversal of the earlier pair e_1 and e'_1.
2. If a chosen path of minimal VECA cost (step 3 or 3') from vertex v to vertex w does not contain pairs of edges traversed $k+1$ or $k+2$ times, then this path traversal leaves the algorithm within the same $ZC(w)$.
3. If edge $e = (v, w)$ from MaxST is included in the selected path of minimal VECA cost with the direction towards the root (starting vertex), it guarantees that all vertices below v in MaxST do not contain emanating unexplored edges, otherwise the path from v to that vertices would have lower VECA cost than $cost(e)$.

Thus, the algorithm is initially located in a zero-component of the starting vertex. It expands the current zero-component by exploring new edges, or possibly splits it in several zero-components after traversing pairs of edges for the kth time and assigning them positive VECA cost. However, unless a pair of edges is traversed for the $k+1$st or $k+2$nd time, the algorithm is still in the same zero-component. It is exactly the path containing pairs of edges traversed for the $k+1$st or $k+2$nd time that relocates the algorithm from one zero-component to another one, after which the algorithm stays in the zero-component of the last vertex of the path, until the next path of minimal VECA cost is selected which also contains pairs of edges to be traversed for the $k+1$st or $k+2$nd time. MaxST has monotonically decreasing VECA costs from the root (starting vertex) towards leaves. This fact guarantees that whenever a pair of MaxST-edges is traversed for the $k+2$nd time, there is no need in re-traversing it again, because all vertices reachable through this pair of edges do not have emanating unexplored (or, alternatively, zero cost) edges.

Thus, if there are no goal states, Basic-VECA explores all the edges of G, otherwise Basic-VECA stop successfully when it hits a goal state. Each pair of twin edge is traversed at most $k+2$ times, which implies the worst-case complexity of at most $(k/2+1) \times weight(G)$. ■

A larger k allows the exploitation algorithm to maintain its original behavior longer, whereas a smaller k forces it earlier to explore the state space more. The smaller the value of k, the better the performance guarantee of Basic-VECA. If $k = 0$, for example, Basic-VECA severely restricts the freedom of the exploitation algorithm and behaves like chronological backtracking. In this case, it traverses every edge at most once (for a total cost of $weight$), no matter how misleading its heuristic knowledge is or how bad the choices of the exploitation algorithm are. No uninformed goal-directed exploration algorithm can do better in the worst case even if an edge traversal also identifies the twin of the edge. However, if the heuristic values are not misleading, a small value of k can force the exploitation algorithm to explore the graph unnecessarily. Thus, a stronger performance guarantee might come at the expense of a decrease in average-case performance. The experiments in the section on "Experimental Results" address this issue.

VECA is very similar to Basic-VECA, see Figure 4. In contrast to Basic-VECA, however, it does not assume that an edge traversal identifies the twin of the edge. This complicates the algorithm somewhat: First, the twin of an edge might not be known when VECA reserves a VECA cost for the pair. This requires an additional amount of bookkeeping. Second, the twin of an edge might not be known when VECA wants to assign it positive VECA cost. In this case, VECA is forced to identify the twin: step 4(e) performs a Eulerian walk on part of the graph, it explores at least all edges adjacent to the vertex that contains the twin (including the twin) and returns to that vertex. This procedure is executed only rarely for larger k, since it is almost never the case that the twin of an edge that has already been executed k times has not yet been explored. Because of this step, though, VECA can potentially traverse any edge one more time than Basic-VECA, which implies the following result:

Corollary 1 *VECA, with even parameter $k \geq 0$, solves any goal-directed exploration problem with a cost of $O(1) \times weight$ (to be precise: with a cost of at most $(k/2+2) \times weight$).*

For $k = 0$, VECA traverses every edge at most twice. Thus, its worst-case performance is at most $2 \times weight$ and equals the worst-case performance of BETA. No uninformed goal-directed exploration algorithm can do better in the worst case if executing an action does not identify its twin.

5 Implementation Since the VECA costs are exponentially decreasing and the precision of numbers on a computer is limited, Basic-VECA and VECA cannot be implemented exactly as described. Instead, we represent paths as lists that contain the current VECA costs of their edges in descending order. All paths

```
Input:
a goal-directed exploration problem,
the value of VECA's parameter $k$ (a non-negative, even integer),
and a heuristic-driven exploitation algorithm (to be used in step 3).

VECA uses four variables for each edge $e$: $count(e)$ keeps track of how often the edge has been executed,
$reserve(e)$ is its reserved VECA cost, $cost(e)$ is its actual VECA cost, and $euler(e)$ remembers whether it has
already been executed as part of Eulerian walk - step 4(e).
1. Set $count(e) := cost(e) := euler(e) := 0$ for all $e \in E$. Set $i := 0$ and $v := v_{start}$.
2. If $v \in GOAL$, then stop successfully.
3. Consider all cycle-less paths starting at $v$ and ending with an *unexplored edge*. Select one with minimal
   VECA cost from these paths, using the heuristic-driven exploitation algorithm to break ties.
4. Consider all edges in the path, one after another. For each edge $e = (v, w)$ in the path, do:
   (a) Traverse $e$.
   (b) Set $count(e) := count(e) + 1$.
   (c) If $count(e) = 1$ and the twin of $e$ is not yet known, then set $i := i + 1$ and afterwards
       $reserve(e) := 2^{-i}$.
   (d) If $count(e) = 1$ and the twin of $e$ is known, then let $e' = (w, v) \in E$ be the twin of
       $e$. If $count(e') = 0$, then set $i := i + 1$ and afterwards $reserve(e) := 2^{-i}$, else set
       $reserve(s, a) := reserve(e')$.
   (e) If $count(e) \geq k$ and the twin of $e$ is not yet known, then perform a Eulerian walk:
       i. Set $v'' := succ(e)$.
       ii. Select an edge $e'' = (v'', w'')$ with $euler(e'') = 0$. If there is no such action, then
           go to step 4(f) (comment: it holds that $v'' = succ(e)$).
       iii. Traverse $e''$.
       iv. Set $euler(e'') := 1$.
       v. Set $v'' := succ(e'')$.
       vi. Go to step 4(e)ii.
   (f) If the twin of $e$ is known, then let $e' = (w, v) \in E$ be the twin of $e$ and do:
       i. If $count(e) + count(e') \geq k$ and $cost(e) = 0$, then set $cost(e) := cost(e') :=$
          $reserve(e)$.
       ii. If $count(e) + count(e') > k$, then set $cost(e) := \infty$.
   (g) Set $v := succ(e)$.
5. Go to step 2.
Alternatively, step 3 can be replaced by:
3' Consider all cycle-less paths starting at $s$ and ending with an edge whose VECA cost is zero. Select one
   with minimal VECA cost from these paths, using the heuristic-driven exploitation algorithm to break
   ties.
```

Figure 4: The VECA Framework

of minimal VECA cost then have the smallest lexicographic order. Since this relationship continues to hold if we replace the VECA costs of the edges with their exponent (for example, we use -3 if the VECA cost of an edge is $1/8 = 2^{-3}$), we can now use small integers instead of exponentially decreasing real values, and steps 3 and 3' can be implemented efficiently using a simple modification of Dijkstra's algorithm in conjunction with priority lists.

6 Experimental Results We augment our theoretical worst-case analysis with an experimental average-case analysis, because the worst-case complexity of an algorithm often does not predict its average-case performance well. The task that we studied was finding goals in mazes. The mazes were constructed by first generating an acyclic maze of size 64×64 and then randomly removing 32 walls. The costs corresponded to the travel distances; the shortest distance between two junctions counted as one unit. We randomly created ten mazes with start location (62,62), goal location (0,0), diameters between 900 and 1000 units, and goal distances of the start state between 650 and 750 units. For every goal-directed exploration algorithm, we performed ten trials in each of the ten mazes (with ties broken randomly). We repeated the experiments for different values of VECA's parameter k and for heuristic values $h(s,a)$ of different quality. The heuristic values were generated by combining the goal distance $gd(s)$ of a state s with its Manhattan distance $mh(s)$, the sum of the x and y distance from s to the goal state. A parameter $t \in [0,1]$ determined how misleading the heuristic values were; a smaller t implied a lower quality:

$$h(s,a) = c(s,a) + t \times gd(succ(s,a)) + (1-t) \times mh(succ(s,a)).$$

Here, we report the results for two heuristic-driven exploitation algorithms, namely AC-A* and Learning Real-Time A* (LRTA*) with look-ahead one [5]. We integrated these algorithms into the VECA framework as follows: AC-A* was used with step 3 of VECA and broke ties among action sequences according to their total cost (see the definition of AC-A*). LRTA* was used with step 3' of VECA and broke ties according to the heuristic value of the last action in the sequence. These ways of integrating AC-A* and LRTA* with VECA are natural extensions of the stand-alone behavior of these algorithms. If the heuristic values are misleading, AC-A* is more efficient than LRTA* (this is to be expected, since AC-A* deliberates more between action executions). As a result, VECA was able to improve the average-case performance of LRTA* more than it could improve the performance of AC-A*.

Figure 5(a) shows the average-case performance of AC-A* with and without VECA (including 95 percent confidence intervals). The x axis shows $100 \times t$, our measure for the quality of the heuristic values, and the y axis shows the average travel distance from the start to the goal, averaged over all 100 trials. All graphs tend to decrease for increasing t, implying that the performance of the algorithms increased with the quality of the heuristic values (as expected). AC-A* without VECA was already efficient and executed each action only a small number of times. Thus, VECA did not change the behavior of AC-A* when k was large. For example, for $k = 10$, the behavior of AC-A* with VECA (not shown in the figure) was the same as the behavior of AC-A* without VECA. The graphs for $k = 4$ suggest that AC-A* with VECA outperforms AC-A* without VECA if the heuristic values are of bad quality and that it remains competitive even for heuristic values of higher quality.

Figure 5(b) shows the average-case performance of LRTA* with and without VECA. As before, the performance of the algorithms increased with the quality of the heuristic values. The graphs show that, most of the time, LRTA* with VECA outperformed or tied with LRTA* without VECA. For misleading heuristic values (small t), LRTA* with VECA worked the better, the smaller the value of VECA's parameter k was. However, for only moderately misleading heuristic values (t between 0.3 and 0.5), a larger value of k achieved better results and LRTA* without VECA even outperformed LRTA* with VECA if k was small. This was the case, because the heuristic values guided the search better and a small value of k forced

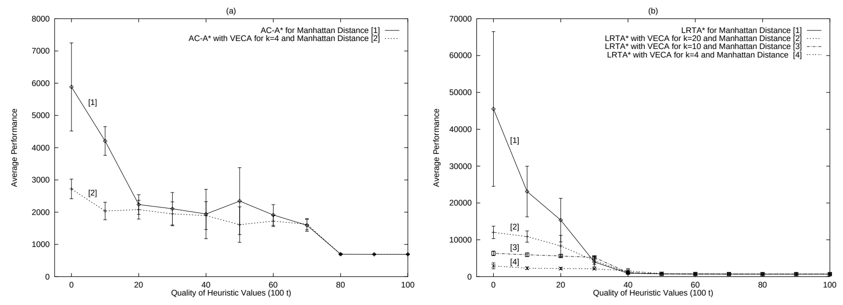

Figure 5: AC-A* and LRTA* with and without VECA

LRTA* to explore the state space too much.

We obtained similar results for both experiments when we generated the heuristic values differently, for example as the combination of the goal distance and the Euclidean distance or the goal distance and the larger coordinate difference between a state and the goal. We also performed experiments with other heuristic-driven exploitation algorithms, such as biased random walks or an algorithm that expands the states in the same order as the A* algorithm. Since both of these algorithms are inefficient to begin with, VECA was able to achieve large performance improvements for misleading heuristic values.

7 Extensions In this paper, we have assumed that a goal-directed exploration algorithm learns only the effect of the executed action, but not the effects of any other actions. However, Basic-VECA and VECA can be used unchanged in environments in which action executions provide more information (such as, for example, complete information about the effects of all actions in some neighborhood of the agent) and Theorem 2 and Corollary 1 continue to hold.

8 Conclusion We introduced an application-independent framework for exploration and heuristic-driven "treasure hunt", called VECA, that addresses a variety of exploration and search problems within the same framework and provides good performance guarantees. VECA can accommodate a wide variety of exploitation strategies and allows to use heuristic knowledge to guide the search towards a goal state. VECA monitors whether the heuristic-driven exploitation algorithm appears to perform poorly on some part of the state space. If so, it forces the exploitation algorithm to explore the state space more. This combines the advantages of pure exploration approaches and heuristic-driven exploitation approaches: VECA is able to utilize heuristic knowledge, but provides a better performance guarantee than previously studied heuristic-driven exploitation algorithms (such as the AC-A* algorithm): VECA's worst-case performance is always linear in the weight of the state space. Thus, while misleading heuristic values do not help it to find a goal state faster, they cannot completely deteriorate its performance either. A parameter of VECA determines when it starts to restrict the choices of the exploitation algorithm. This allows one to trade-off stronger performance guarantees (in case the heuristic knowledge is misleading or because of bad choices of the exploitation algorithm) and more freedom of the exploitation algorithm (in case the quality of the heuristic knowledge is good). In its most stringent form, VECA's worst-case performance is guaranteed to be as good as that of BETA, the best uninformed goal-directed exploration algorithm. Our experiments suggest that VECA's performance guarantee does not greatly deteriorate the average-case performance of many previously studied heuristic-driven exploitation algorithms if they are used in conjunction with VECA; in many cases, VECA even improved their performance.

References

[1] X. Deng and C. Papadimitriou, *Exploring an unknown graph*, in Proceedings of the Symposium on Foundations of Computer Science, 1990, pp. 355–361.

[2] C. Hierholzer, *Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren*, Math. Ann., 6 (1873).

[3] S. Koenig and Y. Smirnov, *Learning graphs with a nearest neighbor approach*, in Proceedings of the Conference on Learning Theory, 1996.

[4] E. Korach, S. Kutten, and S. Moran, *A modular technique for the design of efficient distributed leader finding algorithms*, ACM Transactions on Programming Languages and Systems, 12 (1990).

[5] R. Korf, *Real-time heuristic search*, Artificial Intelligence, 42 (1990), pp. 189–211.

[6] A. Moore and C. Atkeson, *Prioritized sweeping: Reinforcement learning with less data and less time*, Machine Learning, 13 (1993), pp. 103–130.

[7] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Menlo Park, California, 1985.

[8] J. Pemberton and R. Korf, *Incremental path planning on graphs with cycles*, in Proceedings of the AI Planning Systems Conference, 1992, pp. 179–188.

[9] D. Rosenkrantz, R. Stearns, and P. Lewis, *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal of Computing, 6 (1977), pp. 563–581.

[10] Y. Smirnov, S. Koenig, M. Veloso, and R. Simmons, *Efficient goal-directed exploration*, in Proceedings of AAAI, 1996.

[11] A. Stentz, *The focussed D* algorithm for real-time replanning*, in Proceedings of the IJCAI, 1995, pp. 1652–1659.