Random Node Search on Weighted Graphs

by Alejandro Morera Alvarez

1 Globle as a Graph-Theoretic Problem

The search for a fixed, random country can be abstracted as the strategic hunt for a random node on a weighted graph.

Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space, $G = (V, E)$ be a complete, finite graph, that is, $|V| < \infty$ and $E = \{\{u, v\} \mid u \in V, v \in V \setminus \{u\}\}$. We take ξ to be a random variable representing a node chosen at random from some distribution over V. Formally, this is given by a random variable

$$\xi : (\Omega, \mathcal{F}) \to (V, \mathcal{P}(V)).$$

We call ξ the **target** and for $\omega \in \Omega$, an evaluation $\xi(\omega) \in V$ is referred to a **target node**. With a slight abuse of notation, we shall often write ξ instead of $\xi(\omega)$ when referring to the target node. We will typically assume a uniform distribution over V, that is, for all $v \in V$

$$\mathbb{P}(\xi = v) = \frac{1}{|V|},$$

but this is not a necessary assumption.

We now take a copy of the nodes V and construct a random graph $H = (V, F)$, where the edge set F is defined as:

$$F := \{\{\xi, v\} \mid v \in V \setminus \{\xi\}\}.$$

Note that the randomness of H comes from the edges F, as they all connect to the target node ξ.

To both E and the random edge set F, we assign weights. Let $w_0 : E \to [0, \infty)$ be defined by $e \mapsto w_0(e) = w_0(\{u, v\})$ and $\hat{w} : F \to [0, \infty)$ be defined by $f \mapsto \hat{w}(f) = \hat{w}(\{\xi, u\})$. Note that $\hat{w}$ is itself defined over a random set, and is therefore random, too. The model assumes that the weights w_0 are corrupted. The noise is modeled e.g. via an additive Gaussian. More specifically, we fix a small $\sigma > 0$ and set for $e \in E$

$$\tilde{w}(e) := |w_0(e) + Z_e|, \quad \text{where } Z_e \sim \mathcal{N}\left(0, \frac{|P(e)|}{2|V|}\sigma^2\right),$$

where, for $e = \{u, v\}$, we write $|P(e)| := |P(u)| + |P(v)|$ for the cardinality of edges that are physically adjacent to either u or v, see the definition below. Intuitively, the more physically adjacent neighbors an edge has, the more uncertain the measurement of the distance between each of its nodes will be. The corruption models the fact that, though we know precisely how far two nodes are, as humans we are bad at estimating how this distance actually looks like with the naked eye.

More generally, one can take $\tilde{w}_0(e)$ to have some general distribution over $[0, \infty)$ centered at the uncorrupted values:

$$\forall e \in E : \mathbb{E}[\tilde{w}_0(e)] = w_0(e).$$

The weights w_0 are meant to model physical proximity between the nodes, and we assume that the uncorrupted weights coincide in their common domain, that is, for all $v \in V$, we have

$$w_0(\{\xi, v\}) = \hat{w}(\{\xi, v\}). \tag{1}$$

We do not assume that the weights are definite. In other words, there exist vertices $u \neq v$ such that $w_0(\{u, v\}) = 0$. We call such vertices **physically adjacent**. The set of physically adjacent nodes to v

$$\{u \in V \mid w_0(\{u, v\}) = 0\}$$

is denoted by $P(v)$. It is important to keep in mind that while all nodes are adjacent in G (since the graph is complete), not all nodes need to be physically adjacent. In fact, in any non-trivial example, not all nodes are physically adjacent. Otherwise, the search boils down to pure luck.

Task: Find ξ by "guessing".

Constraints: Not everything is known. The known data is:

- the vertices V,
- the fact that G is complete,
- the corrupted weights $\tilde{w}_0$.

In particular, we see that if we knew the uncorrupted weights w_0, then, by (1), we would know all the values of $\hat{w}$, since $F \subseteq E$. Nonetheless, even if we knew the uncorrupted data, this still does not solve the problem of finding ξ.

The true weights relative to ξ, that is, the values of $\hat{w}(\{\xi, v\})$ for $v \in V$ are uncovered by guessing. "Guessing" is defined as choosing some $v \in V$ which we think might be ξ. Once we guess, we "reveal" information about the weights $\hat{w}(\{\xi, v\})$. We will formalize these concepts in the section that follows.

The question is:

How do we choose an algorithm that minimizes the amount of "guesses" to find ξ?

One problem is the non-definiteness of w_0: indeed, if we guess a physically adjacent node v of ξ, then, we are left to explore all neighbors of v until we land at ξ. This means that in the worst case, given that we choose a physically adjacent neighbor v to ξ, then we must check $|P(v)|$ amount of vertices. Therefore, leaving luck aside, the question boils down to finding $P(\xi)$. Once this is done, the rest of the game is to explore this set one by one.

But the question still is: how do we find $P(\xi)$ or, even better, ξ itself, most efficiently?

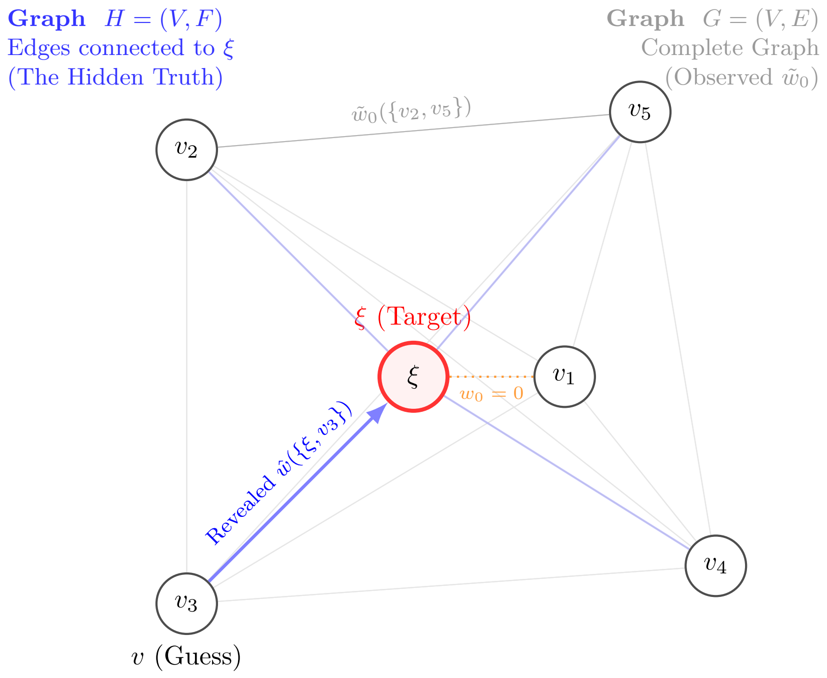

Figure 1: Visualization of the graphs G and H. The faint mesh represents the complete graph G. The blue links represent the hidden graph H centered on ξ. Note that v_1 belongs to the (random) set $P(\xi)$.

2 Algorithmic Approach to Target Identification

We continue with the notation established above. Recall that the task is to find the target node ξ given only the vertices V, the fact that $G = (V, E)$ is complete, and the corrupted weights $\tilde{w}_0$. We now develop an algorithmic framework to minimize the expected number of guesses.

We first formalize the notion of "guessing".

Definition 2.1 A **guess** is defined as a function $\mathcal{G} : V \to \{0, 1\} \times [0, \infty)$ given by

$$\mathcal{G}(v) = \begin{cases} (1, 0) & \text{if } v = \xi, \\ (0, \hat{w}(\{\xi, v\})) & \text{if } v \neq \xi. \end{cases}$$

In other words, querying $\mathcal{G}(v)$ reveals whether $v = \xi$ (first component), and if not, reveals the true weight $\hat{w}(\{\xi, v\})$ (second component). We denote by $\mathcal{G}_1(v)$ and $\mathcal{G}_2(v)$ the first and second components, respectively.

Definition 2.2 (Guessing Strategy) A **guessing strategy** is a sequence of measurable functions $(S_k)_{k \geq 1}$ where

$$S_k : V^{k-1} \times [0, \infty)^{k-1} \to V$$

such that $S_k(v_1, \ldots, v_{k-1}, r_1, \ldots, r_{k-1})$ denotes the k-th guess given the history of previous guesses $(v_1, \ldots, v_{k-1})$ and their revealed weights $(r_1, \ldots, r_{k-1})$.

Definition 2.3 (Stopping Time) For a guessing strategy $S = (S_k)_{k \geq 1}$, define the **stopping time**

$$\tau_S := \inf \{k \geq 1 \mid \mathcal{G}_1(S_k) = 1\} = \inf \{k \geq 1 \mid S_k = \xi\}.$$

This represents the number of guesses required to find ξ. The goal is to find a strategy S^* that minimizes $\mathbb{E}[\tau_{S^*}]$.

2.1 Weight-Based Likelihood Estimation

Given the corrupted weights $\tilde{w}_0$ and the noise model, we can compute posterior probabilities. For each $v \in V$, define the **prior likelihood** that $v = \xi$ based solely on the corrupted weights.

Assuming the additive Gaussian noise model, i.e., $\tilde{w}_0(e) = w_0(e) + Z_e$ with $Z_e \sim \mathcal{N}(0, \frac{|V_e|}{2|V|}\sigma^2)$, we define the **consistency score** of a candidate $v \in V$ as follows. If $v = \xi$, then for all $u \in V \setminus \{v\}$, the corrupted weight $\tilde{w}_0(\{u, v\})$ should be centered around the true weight $\hat{w}(\{\xi, u\})$. We measure **consistency** via

$$C(v) := \sum_{u \in V \setminus \{v\}} \left(\frac{\tilde{w}_0(\{u, v\}) - \bar{w}(v)}{\sigma \sqrt{|V_{\{u,v\}}|/(2|V|)}} \right)^2, \tag{2}$$

where $\bar{w}(v) := \frac{1}{|V|-1} \sum_{u \in V \setminus \{v\}} \tilde{w}_0(\{u, v\})$ is the average corrupted weight incident to v.

Remark 2.4 The intuition behind (2) is that if $v = \xi$, the weights $\{\tilde{w}_0(\{u, v\})\}_{u \neq v}$ should exhibit a pattern consistent with being centered around the true (uncorrupted) weights emanating from the target. Nodes with low consistency scores are more likely to be ξ.

2.2 Handling Physical Adjacency

The non-definiteness of w_0 introduces the complication that guessing a node $v \in P(\xi)$ (i.e., a node physically adjacent to ξ) reveals $\hat{w}(\{\xi, v\}) = 0$, which does not uniquely identify ξ. Define the **zero-weight neighborhood** of v as

$$N_0(v) := \{u \in V \mid \tilde{w}_0(\{u, v\}) \leq \epsilon\}$$

for some threshold $\epsilon > 0$ chosen to account for noise. If $v \in P(\xi)$, then $\xi \in N_0(v)$ with high probability.

Proposition 2.5 Let $v \in V$ be a guess such that $\mathcal{G}_2(v) = 0$. Then $v \in P(\xi)$ and $\xi \in P(v)$. In particular, $\xi \in N_0(v)$ almost surely as $\sigma \to 0$.

2.3 Adaptive Target Identification (ATI) Algorithm

We now present an algorithm that adaptively refines the search space using revealed weights.

Algorithm 1 Adaptive Target Identification (ATI)

```
Require: $V$, corrupted weights $\tilde{w}_0$, noise parameter $\sigma$, threshold $\epsilon$
Ensure: Target node $\xi$
 1: Initialize: $A \leftarrow V$                                  ▷ Active candidate set
 2: Initialize: $\mathcal{H} \leftarrow \emptyset$                 ▷ History of (guess, revealed weight) pairs
 3: while $|A| > 1$ do
 4:     Compute consistency scores $\{C(v)\}_{v \in A}$ via (2)
 5:     Compute posterior probabilities:
            $$p(v) \propto \exp\left(-\frac{C(v)}{2}\right) \cdot \mathbb{P}(\xi = v) \quad \text{for } v \in A$$
 6:     Select $v^* \leftarrow \arg\max_{v \in A} p(v)$
 7:     Query $\mathcal{G}(v^*)$
 8:     if $\mathcal{G}_1(v^*) = 1$ then
 9:         return $v^*$                                          ▷ Found $\xi$
10:     end if
11:     $r^* \leftarrow \mathcal{G}_2(v^*)$                       ▷ Revealed weight $\hat{w}(\{\xi, v^*\})$
12:     $\mathcal{H} \leftarrow \mathcal{H} \cup \{(v^*, r^*)\}$
13:     if $r^* \le \epsilon$ then                                 ▷ Physically adjacent
14:         $A \leftarrow N_0(v^*) \setminus \{v^*\}$
15:     else
16:         Update $A$ by removing nodes inconsistent with $r^*$:
                $$A \leftarrow \{u \in A \setminus \{v^*\} \mid |\tilde{w}_0(\{u, v^*\}) - r^*| \le \delta(\sigma)\}$$
17:     end if
18: end while
19: return the unique element of $A$
```

Here, $\delta(\sigma) > 0$ is a confidence band that depends on σ; a natural choice is $\delta(\sigma) = 3\sigma\sqrt{(2|V|)^{-1} \cdot \max_{e \in E} |V_e|}$.

2.4 Refinement via Revealed Weights

The key insight is that each guess $v \neq \xi$ reveals $\hat{w}(\{\xi, v\})$, which constrains the location of ξ. For any candidate $u \in A$, if $u = \xi$, then by (1), we must have

$$w_0(\{u, v\}) = \hat{w}(\{\xi, v\}) = r^*.$$

Since we only observe $\tilde{w}_0$, we keep u in the active set A if and only if

$$|\tilde{w}_0(\{u, v\}) - r^*| \le \delta(\sigma),$$

i.e., the corrupted weight is within a noise-tolerant band of the revealed true weight.

Lemma 2.6 (Candidate Elimination) Let $v \in V \setminus \{\xi\}$ be a guess with revealed weight $r^* = \hat{w}(\{\xi, v\})$. Then, for any $u \in V \setminus \{v, \xi\}$,

$$\mathbb{P}\left(|\tilde{w}_0(\{u,v\}) - r^*| > \delta(\sigma)\right) \geq 1 - 2\Phi\left(-\frac{\delta(\sigma) - |w_0(\{u,v\}) - r^*|}{\sigma\sqrt{|V_{\{u,v\}}|/(2|V|)}}\right),$$

where Φ is the standard normal CDF. In particular, if $|w_0(\{u,v\}) - r^*|$ is large relative to σ, then u is eliminated with high probability.

2.5 Complexity Analysis

Theorem 2.7 (ATI is Logarithmic in $|V|$)
Let $n = |V|$ and assume ξ is uniformly distributed over V. Then the expected number of guesses under Algorithm 1 satisfies

$$\mathbb{E}[\tau_{\text{ATI}}] \leq \log_2(n) + \max_{v \in V} |P(v)| + O\left(\frac{\sigma}{\Delta_{\min}}\right), \tag{3}$$

where $\Delta_{\min} := \min\{|w_0(e) - w_0(e')| : e \neq e', w_0(e) \neq w_0(e')\}$ is the minimum weight gap.

Proof Sketch. Each guess $v \neq \xi$ with $\hat{w}(\{\xi, v\}) > \epsilon$ eliminates approximately half of the remaining candidates in expectation (under suitable distributional assumptions on w_0), yielding the $\log_2(n)$ term. If we guess a node in $P(\xi)$, we must search through $P(\xi)$, contributing at most $\max_{v \in V} |P(v)|$ additional guesses. The error term $O(\sigma/\Delta_{\min})$ accounts for misclassification due to noise when weight gaps are small. □

2.6 Optimality Considerations

Theorem 2.8 (Lower Bound) For any guessing strategy S,

$$\mathbb{E}[\tau_S] \geq \frac{\log n}{H(\mathbf{p})} + \mathbb{E}[|P(\xi)|] - 1,$$

where $H(\mathbf{p}) = -\sum_{v \in V} \mathbb{P}(\xi = v) \log \mathbb{P}(\xi = v)$ is the entropy of the target distribution.

Remark 2.9 When ξ is uniform, $H(\mathbf{p}) = \log n$, and the lower bound becomes $1 + \mathbb{E}[|P(\xi)|] - 1 = \mathbb{E}[|P(\xi)|]$. This shows that the physical adjacency structure fundamentally limits the best achievable performance. The $\log_2(n)$ term in (3) reflects the information-theoretic cost of locating $P(\xi)$ within V. Observe that the size of $P(\xi)$ depends on the weight function $\hat{w}$ and not on the structure of the graph itself.

2.7 Cluster-Aware Target Identification (CATI) Algorithm

When the physical adjacency structure exhibits clustering (i.e., nodes in $P(v)$ are themselves close to each other), we can exploit this to improve performance. Define the **physical adjacency graph** $G_P = (V, E_P)$ where

$$E_P := \{\{u,v\} \in E \mid w_0(\{u,v\}) = 0\}.$$

Let $\{C_1, \ldots, C_m\}$ be the connected components of G_P.

Algorithm 2 Cluster-Aware Target Identification (CATI)

```
Require: $V$, corrupted weights $\tilde{w}_0$, noise parameter $\sigma$
Ensure: Target node $\xi$
1: Estimate clusters $\left\{\hat{C}_1, \dots, \hat{C}_{\hat{m}}\right\}$ from $\tilde{w}_0$ using hierarchical clustering with threshold $\epsilon$
2: Compute cluster representatives $\{r_1, \dots, r_{\hat{m}}\}$ where $r_i \in \hat{C}_i$
3: Apply Algorithm 1 to $\{r_1, \dots, r_{\hat{m}}\}$ to find cluster $\hat{C}^*$ containing $\xi$
4: Apply Algorithm 1 within $\hat{C}^*$ to find $\xi$
5: return $\xi$
```

Theorem 2.10 (CATI is Logarithmic in G_P-Components)
Under Algorithm 2, if the cluster estimation is correct, then

$$\mathbb{E}[\tau_{\text{CATI}}] \leq \log_2(m) + \mathbb{E}[|C_\xi|], \tag{4}$$

where C_ξ denotes the cluster containing ξ and m is the number of clusters.

In short, therefore, an optimal algorithm for finding ξ proceeds in two phases:

1. **Localization Phase:** Use the corrupted weights $\tilde{w}_0$ to compute consistency scores and iteratively guess high-probability candidates. Each guess refines the candidate set using the revealed true weight.
2. **Exploration Phase:** Once a physically adjacent node is found (i.e., $\hat{w}(\{\xi, v\}) = 0$), exhaustively search the zero-weight neighborhood $N_0(v)$.

The key insight is that revealed weights $\hat{w}(\{\xi, v\})$ provide *exact* information about the true edge weights incident to ξ, allowing rapid elimination of inconsistent candidates despite only having access to corrupted weights $\tilde{w}_0$.

3 ATI Algorithm in Action

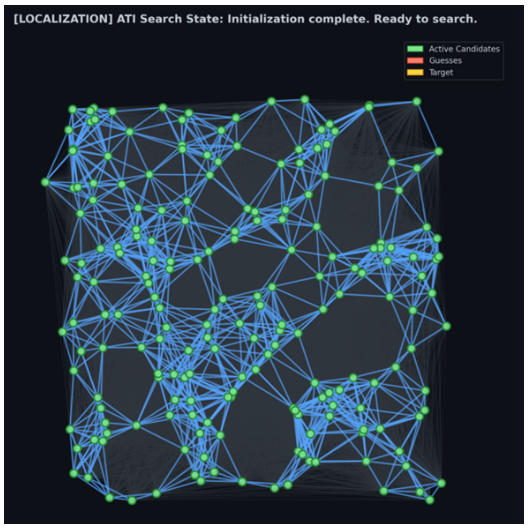
[LOCALIZATION] ATI Search State: Initialization complete. Ready to search.
🟩 Active Candidates
🟥 Guesses
🟨 Target

Figure 2: ATI Algorithm: Uninitialized. The graph G consists of 200 nodes.

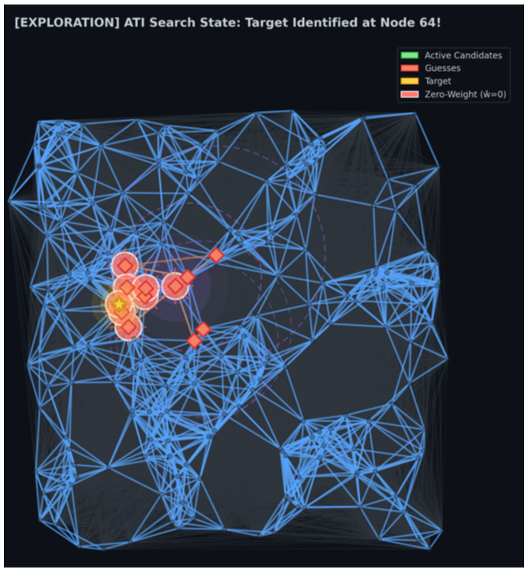
[EXPLORATION] ATI Search State: Target Identified at Node 64!
🟩 Active Candidates
🟥 Guesses
🟨 Target
🟥 Zero-Weight (ŵ=0)

Figure 3: ATI Algorithm: Finalized Search for $P(\xi)$ in 4 Steps and for ξ in 12 Steps.

Step	Node ID	Weight	Type
12	64	0.0000	**FOUND**
11	110	0.0000	EXPLORE
10	190	0.0000	EXPLORE
9	53	0.0000	EXPLORE
8	127	0.0000	EXPLORE
7	97	0.0000	EXPLORE
6	79	0.1678	EXPLORE
5	187	0.0000	EXPLORE
4	41	0.2476	EXPLORE
3	81	0.0000	SEARCH
2	87	0.1903	SEARCH
1	150	0.2000	SEARCH

Table 1: Sample of the ATI Algorithm found an element of $P(\xi)$ on the fourth step. The rest of the steps explores this set in a step-by-step basis.

A Code: Implementation of the ATI Logic

```python
"""
Globle: Mathematical Framework for Target Identification on Complete Graphs
Based on: "Globle over General Graphs" - Algorithmic Approach to Target Identification

Improved Algorithm with insights from the paper:
1. Zero-weight detection -- target is physically adjacent
2. Information-theoretic candidate selection
3. Physical cluster exploration phase
4. Multi-constraint triangulation
"""
import numpy as np
from scipy.spatial.distance import pdist, squareform
from typing import Optional


class GlobleEnvironment:
    """
    Models the probabilistic environment from the paper.

    G = (V, E) is a COMPLETE graph where E = {{u,v} | u,v     V, u     v}

    Key objects:
    -  w  : E     [0,   ) - True underlying weights (physical proximity)
    -  w    : Corrupted weights with Gaussian noise
    -    : Target node (random variable, uniform over V)
    -    : F     [0,   ) - Weights from target to all other nodes

    Physical adjacency:  w  ({u,v}) = 0 means u,v are "physically adjacent"
    """

    def __init__(self, num_nodes: int, sigma: float, physical_threshold: float = 0.15):
        self.N = num_nodes
        self.sigma = sigma
        self.nodes = list(range(num_nodes))
        self.physical_threshold = physical_threshold

        # Generate node positions in 2D unit square (defines underlying metric)
        self.positions = np.random.rand(num_nodes, 2)

        # Compute pairwise Euclidean distances (true weights  w  )
        dists = squareform(pdist(self.positions))

        # Apply physical adjacency threshold:  w  ({u,v}) = 0 if too close
        # Paper: "There exist vertices  u v  such that  w  ({u,v})=0. We call such vertices physically adjacent."
        self.w0 = np.where(dists < physical_threshold, 0.0, dists)
        np.fill_diagonal(self.w0, 0.0)

        # Generate corrupted weights  w
        # Paper:  w  (e) :=  w  (e) +  Z  , where Z ~ N(0, (| V  |/|V|)      )
        Ve_max = 2 * self.N - 3
        variance = (Ve_max / self.N) * (self.sigma ** 2)
        noise_std = np.sqrt(variance)

        # Symmetric noise matrix
        noise = np.random.normal(0, noise_std, size=(self.N, self.N))
        noise = (noise + noise.T) / 2

        self.w_tilde = self.w0 + noise
        self.w_tilde = np.maximum(self.w_tilde, 0)  # Weights are non-negative
        np.fill_diagonal(self.w_tilde, 0.0)

        # Select target    uniformly at random
        self.target_idx = np.random.choice(self.nodes)

        # Precompute physical adjacency sets P(v)
        # Paper: P(v) = {{u,v}     E |  w  ({u,v}) = 0}
        self.physical_adj = {
            v: set(u for u in self.nodes if u != v and self.w0[v, u] == 0)
            for v in self.nodes
        }

        # Precompute physical clusters (connected components of physical adjacency graph)
        self._compute_physical_clusters()

    def _compute_physical_clusters(self):
        """Find connected components in the physical adjacency graph."""
        visited = set()
        self.clusters = []  # List of sets
        self.node_to_cluster = {}  # node -> cluster index

        for start in self.nodes:
            if start in visited:
                continue

            # BFS to find connected component
            cluster = set()
            queue = [start]
            while queue:
                node = queue.pop(0)
                if node in visited:
                    continue
                visited.add(node)
                cluster.add(node)
                for neighbor in self.physical_adj[node]:
                    if neighbor not in visited:
                        queue.append(neighbor)

            cluster_idx = len(self.clusters)
            self.clusters.append(cluster)
            for node in cluster:
                self.node_to_cluster[node] = cluster_idx

    def query_oracle(self, v: int) -> tuple[bool, float]:
        """
        The Guess Oracle G(v) from Definition 1.1.1:

        G(v) = {
            (1, 0)           if v =
            (0,   ({  , v}))   if v
        }

        Returns (is_target, revealed_weight)
        """
        if v == self.target_idx:
            return (True, 0.0)
        else:
            return (False, self.w0[self.target_idx, v])

    def get_confidence_band(self) -> float:
        """
        Compute   (  ) - the confidence band for filtering candidates.
        Paper:   (  ) = 3     (|V|         max_{ e E } | V  |)
        """
        Ve_max = 2 * self.N - 3
        return 3 * self.sigma * np.sqrt(Ve_max / self.N)

    def get_physical_neighborhood(self, v: int) -> set:
        """Get P(v) - all nodes physically adjacent to v."""
        return self.physical_adj[v]

    def get_cluster(self, v: int) -> set:
        """Get the physical cluster containing v."""
        return self.clusters[self.node_to_cluster[v]]


class ATIAlgorithm:
    """
    Improved Algorithm 1: Adaptive Target Identification (ATI)

    Key Improvements:
    1. ZERO-WEIGHT EXPLOITATION: When   ({  ,v}) = 0, target is in P(v)
           Immediately restrict to physical neighborhood

    2. INFORMATION-THEORETIC SELECTION: Pick candidates that maximize
       expected information gain (split the candidate set most evenly)

    3. EXPLORATION PHASE: When in a physical cluster, exhaustively search

    4. MULTI-CONSTRAINT TRIANGULATION: Use all revealed weights jointly

    Phases (from paper section 1.1.8):
    - Localization Phase: Use corrupted weights to find P(  )
    - Exploration Phase: Once physically adjacent, search the cluster
    """

    def __init__(self, env: GlobleEnvironment):
        self.env = env
        self.active_set = set(env.nodes)
        self.history = []
        self.revealed_constraints = []
        self.found = False
        self.status_message = "Initialization complete. Ready to search."

        self.epsilon = 1e-9  # True zero detection (not threshold!)
        self.delta = env.get_confidence_band()

        # Phase tracking
        self.phase = "LOCALIZATION"  # or "EXPLORATION"
        self.physical_neighbor_found: Optional[int] = None  # Node where we found w=0
        self.exploration_cluster: Optional[set] = None  # Cluster to search

        # Statistics
        self.stats = {
            'zero_weight_detections': 0,
            'candidates_eliminated': 0,
            'phase_switches': 0
        }

    def _compute_consistency_scores(self) -> dict[int, float]:
        """
        Compute consistency score C(v) for each candidate v     A.

        C(v) =  _ {(v*, r*)     H} (( w    ({v, v*}) - r*) /      )

        Lower C(v) = more consistent = more likely to be target.
        """
        scores = {}

        Ve_max = 2 * self.env.N - 3
        sigma_e = self.env.sigma * np.sqrt(Ve_max / self.env.N)
        if sigma_e < 1e-10:
            sigma_e = 1e-10

        for v in self.active_set:
            if len(self.revealed_constraints) == 0:
                # No constraints yet - use variance heuristic
                incident = self.env.w_tilde[v, :]
                incident = np.delete(incident, v)
                scores[v] = np.var(incident)
            else:
                # Compute consistency with ALL revealed weights
                total = 0.0
                for (guessed_node, revealed_weight) in self.revealed_constraints:
                    w_tilde_v_guess = self.env.w_tilde[v, guessed_node]
                    diff = w_tilde_v_guess - revealed_weight
                    total += (diff / sigma_e) ** 2
                scores[v] = total

        return scores

    def _select_informative_candidate(self) -> int:
        """
        Select the candidate that would provide maximum information gain.

        Improvement: Instead of just picking highest posterior, consider
        which guess would best split the remaining candidate set.

        Strategy: Pick a candidate near the "center" of the active set,
        as this maximizes expected candidate elimination.
        """
        if len(self.active_set) <= 2:
            # Just use consistency for small sets
            scores = self._compute_consistency_scores()
            min_score = min(scores.values())
            posteriors = {v: np.exp(-(scores[v] - min_score) / 2) for v in scores}
            return max(posteriors, key=posteriors.get)

        # Compute centroid of active candidates
        active_list = list(self.active_set)
        positions = self.env.positions[active_list]
        centroid = np.mean(positions, axis=0)

        # Find candidate closest to centroid (would split set most evenly)
        distances_to_centroid = np.linalg.norm(positions - centroid, axis=1)

        # Combine with consistency scores
        scores = self._compute_consistency_scores()
        min_score = min(scores.values())

        # Composite score: low consistency + close to centroid
        composite = {}
        for i, v in enumerate(active_list):
            consistency_weight = np.exp(-(scores[v] - min_score) / 2)
            # Prefer nodes closer to centroid (invert distance)
            centrality = 1.0 / (1.0 + distances_to_centroid[i])
            composite[v] = consistency_weight * (1 + 0.3 * centrality)

        return max(composite, key=composite.get)

    def _refine_candidates_with_constraint(self, guessed_node: int, revealed_weight: float):
        """
        Use revealed weight to eliminate inconsistent candidates.

        Key insight from paper: | w    ({u, v*}) - r*|       (   )

        If the revealed weight is EXACTLY zero, we know target is
        physically adjacent to guessed_node.
        """
        initial_size = len(self.active_set)

        if revealed_weight < self.epsilon:
            #


            # ZERO WEIGHT DETECTED: Target is physically adjacent to v*
            # Paper: "if we guess a physically adjacent node v of    ,
            # then we are left to explore all neighbors of v"
            #


            self.stats['zero_weight_detections'] += 1

            # Get physical neighborhood of guessed node
            physical_neighbors = self.env.get_physical_neighborhood(guessed_node)

            # Target MUST be in P(guessed_node)
            # Intersect with current active set
            self.active_set = self.active_set & physical_neighbors

            # Switch to exploration phase
            if self.phase != "EXPLORATION":
                self.phase = "EXPLORATION"
                self.physical_neighbor_found = guessed_node
                self.exploration_cluster = self.env.get_cluster(guessed_node)
                self.stats['phase_switches'] += 1

            self.status_message = f"Node {guessed_node}:    = 0! Target in P({guessed_node}). Exploring {len(self.active_set)} neighbors."
        else:
            # Standard refinement with all constraints
            new_active = set()
            for u in self.active_set:
                if u == guessed_node:
                    continue

                # Check consistency with THIS constraint
                w_tilde_uv = self.env.w_tilde[u, guessed_node]
                if abs(w_tilde_uv - revealed_weight) <= self.delta:
                    # Also check consistency with ALL previous constraints
                    is_consistent = True
                    for (prev_node, prev_weight) in self.revealed_constraints[:-1]:
                        w_tilde_u_prev = self.env.w_tilde[u, prev_node]
                        if abs(w_tilde_u_prev - prev_weight) > self.delta:
                            is_consistent = False
                            break

                    if is_consistent:
                        new_active.add(u)

            self.active_set = new_active
            self.status_message = f"Node {guessed_node}:    = {revealed_weight:.4f}. Eliminating inconsistent candidates."

        eliminated = initial_size - len(self.active_set)
```

```python
        self.stats['candidates_eliminated'] += eliminated

    def step(self) -> None:
        """Execute one step of the improved ATI algorithm."""
        if self.found or len(self.active_set) == 0:
            return

        if len(self.active_set) == 1:
            # Only one candidate left - must be target
            v_star = list(self.active_set)[0]
            is_target, weight = self.env.query_oracle(v_star)
            self.revealed_constraints.append((v_star, weight))
            self.history.append({'guess': v_star, 'weight': weight, 'type': 'FOUND'})
            self.found = True
            self.status_message = f"Target Identified at Node {v_star}!"
            return

        # Select next candidate to query
        if self.phase == "EXPLORATION":
            # In exploration phase: use consistency to pick from physical cluster
            scores = self._compute_consistency_scores()
            min_score = min(scores.values()) if scores else 0
            posteriors = {v: np.exp(-(scores[v] - min_score) / 2) for v in scores}
            v_star = max(posteriors, key=posteriors.get) if posteriors else list(self.active_set)[0]
        else:
            # In localization phase: use information-theoretic selection
            v_star = self._select_informative_candidate()

        # Query oracle
        is_target, revealed_weight = self.env.query_oracle(v_star)
        self.revealed_constraints.append((v_star, revealed_weight))

        if is_target:
            self.found = True
            self.history.append({'guess': v_star, 'weight': 0.0, 'type': 'FOUND'})
            self.status_message = f"Target Identified at Node {v_star}!"
            return

        # Record guess
        guess_type = 'EXPLORE' if self.phase == "EXPLORATION" else 'SEARCH'
        self.history.append({'guess': v_star, 'weight': revealed_weight, 'type': guess_type})

        # Refine candidates using revealed weight
        self._refine_candidates_with_constraint(v_star, revealed_weight)

        if len(self.active_set) == 0:
            self.status_message = "Error: All candidates eliminated. Noise too high?"

    def get_phase_info(self) -> str:
        """Get human-readable phase information."""
        if self.phase == "EXPLORATION":
            return f"EXPLORATION (searching P({self.physical_neighbor_found}))"
        return "LOCALIZATION (searching globally)"
```

Listing 1: ATI Algorithm Logic