Technical Companion to

A Generalization of the Shortest Path Problem to Graphs with Multiple Edge-Cost Estimates

A Self-Contained Guide for CS Undergraduates

Companion to the paper by Eyal Weiss, Ariel Felner, and Gal A. Kaminka
(ECAI 2023)

Abstract

This document explains every definition, equation, theorem, algorithm, and technical concept in the paper "A Generalization of the Shortest Path Problem to Graphs with Multiple Edge-Cost Estimates" in detail, assuming only undergraduate-level knowledge of graph theory, Dijkstra's algorithm, and basic algorithm analysis. No prior knowledge of multi-cost estimation, anytime algorithms, or the specific search algorithms discussed in the paper is required. Each equation and definition is accompanied by (1) a plain-English description of what it computes or states, (2) a symbol-by-symbol breakdown, (3) an intuitive explanation of why the concept makes sense, and (4) a concrete numerical example where helpful.

Contents

1 Background: Shortest Paths and Why Edge Costs Might Be Uncertain

1.1 The Standard Shortest Path Problem

The **shortest path problem** is one of the most fundamental problems in computer science. Given a *directed, weighted graph*, the goal is to find a path from a start vertex to a goal vertex whose total weight (cost) is minimized.

Key Idea

A **weighted directed graph** is a tuple (V, E, c) where V is a set of vertices, $E \subseteq V \times V$ is a set of directed edges, and $c : E \to \mathbb{R}^+$ is a *cost function* that assigns a non-negative real number to each edge. A **path** from vertex v_i to vertex v_j is a sequence of edges that connects them, and the **cost of a path** is the sum of the costs of all its edges.

Formally, a path $p = \langle e_1, \ldots, e_n \rangle$ from v_i to v_j is a sequence of edges $e_k = (v_{q_k}, v_{q_{k+1}})$ for $k \in [1, n]$, with $v_i = v_{q_1}$ and $v_j = v_{q_{n+1}}$. The cost of this path is:

$$c(p) := \sum_{k=1}^{n} c(e_k)$$

Symbol-by-symbol breakdown:

- p: a path, represented as a sequence of edges.
- e_k: the k-th edge in the path.
- $c(e_k) \in \mathbb{R}^+$: the cost (weight) of the k-th edge; always non-negative.
- $\sum_{k=1}^{n}$: sum over all edges in the path.
- $c(p)$: the total cost of the path—just the sum of all individual edge costs.

The **Goal-Directed Single-Source Shortest Path** (GDS^3P) problem asks: given a start vertex v_s and a set of goal vertices $V_g \subset V$, find a path π from v_s to some $v \in V_g$ such that $c(\pi)$ is minimized. The optimal cost is denoted C^*.

Numerical Example

Consider a graph with vertices $\{A, B, C, D\}$ and edges with costs:

Edge	Cost $c(e)$
(A, B)	3
(A, C)	1
(B, D)	2
(C, B)	1
(C, D)	5

Start: A. Goal: D. There are three paths from A to D:

- $\langle (A, B), (B, D) \rangle$: cost $= 3 + 2 = 5$.
- $\langle (A, C), (C, D) \rangle$: cost $= 1 + 5 = 6$.
- $\langle (A, C), (C, B), (B, D) \rangle$: cost $= 1 + 1 + 2 = 4$.

The shortest path is $A \to C \to B \to D$ with $C^* = 4$.
Standard algorithms like **Dijkstra's algorithm** and **Uniform-Cost Search (UCS)** solve this by always expanding the vertex with the smallest accumulated cost so far.

1.2 When Edge Costs Are Expensive to Compute

Standard shortest path algorithms assume that looking up the cost $c(e)$ of an edge takes negligible time. But in many real-world settings, this assumption fails.

The Core Motivation

What if computing the exact cost of an edge is *expensive*? For example:

- Querying a remote server for real-time traffic data on a road segment.
- Running an expensive simulation to determine the cost of traversing a terrain cell in robotics.
- Calling an external planner to compute the cost of an action in AI planning.

In these settings, the time spent computing edge weights can dominate the total search time.

The paper decomposes the total search runtime T as:

$$T = T_w + T_v = \tau_w \times w + \tau_v \times v \qquad \text{(Eq. 1)}$$

What it computes: The total runtime is the sum of time spent on edge weight computation (T_w) and time spent on vertex (search) operations (T_v).

Symbol-by-symbol breakdown:

- T: total search runtime.
- T_w: total time spent computing edge weights.
- T_v: total time spent on vertex operations (expanding nodes, managing the priority queue, etc.).
- τ_w: average time to compute *one* edge weight.
- w: total number of edge weight computations performed during the search.
- τ_v: average time per vertex operation.
- v: total number of vertex operations.

The paper then observes that instead of computing each edge weight once (at full accuracy), we can *estimate* each edge weight multiple times at increasing levels of accuracy:

$$T_w = \tau_{w_1} \times w_1 + \tau_{w_2} \times w_2 + \cdots + \tau_{w_k} \times w_k \qquad \text{(Eq. 2)}$$

with $\tau_{w_1} < \tau_{w_2} < \cdots < \tau_{w_k}$, and possibly $w_1 + w_2 + \cdots + w_k > w$.

What this says: Instead of one expensive computation per edge, we use k *estimators* of increasing accuracy and cost. The cheapest estimator gives a rough bound quickly; more expensive estimators tighten the bound. By using cheap estimators on many edges and expensive estimators only on the edges that matter, we can reduce T_w dramatically—even though we may apply estimators more times in total.

Numerical Example

Suppose you are routing a car from Tel Aviv to Haifa. Each road segment's travel time can be estimated at three levels:

1. **Cheap** ($\tau_{w_1} = 1$ ms): Use the road's length and speed limit. Returns bounds $[3, 8]$ minutes for some segment.

2. **Medium** ($\tau_{w_2} = 50$ ms): Query a traffic API for a rough estimate. Returns bounds $[4, 6]$ minutes (tighter).
3. **Expensive** ($\tau_{w_3} = 500$ ms): Run a detailed traffic simulation. Returns bounds $[4.5, 5.0]$ minutes (tightest).

With 1000 road segments in the graph, the standard approach computes every edge at the expensive level: $T_w = 1000 \times 500$ ms $= 500$ seconds. With estimators, perhaps 1000 segments use the cheap estimator, 200 use the medium one, and only 50 use the expensive one: $T_w = 1000 \times 1 + 200 \times 50 + 50 \times 500 = 1000 + 10000 + 25000 = 36000$ ms $= 36$ seconds. That is a $14\times$ speedup!

2 Problem Formulation: Estimated Weighted Digraphs

2.1 Cost Estimators Function (Definition 1)

The paper replaces the standard cost function c with an **estimator-generating function** Θ, which provides multiple estimators for each edge, ordered by increasing runtime and tightening bounds.

Definition 2.1 (Cost Estimators Function)**.** A **cost estimators function** for a set of edges E, denoted as Θ, maps every edge $e \in E$ to a finite and non-empty sequence of weight estimation procedures:

$$\Theta(e) := (\theta_e^1, \theta_e^2, \ldots, \theta_e^{k(e)}), \quad k(e) \in \mathbb{N} \tag{Eq. 3}$$

where estimator θ_e^i, if applied, returns lower and upper bounds (l_e^i, u_e^i) on $c(e)$, such that $0 \leq l_e^i \leq c(e) \leq u_e^i < \infty$. $\Theta(e)$ is ordered by increasing running time of θ_e^i, and the bounds monotonically tighten: $[l_e^j, u_e^j] \subseteq [l_e^i, u_e^i]$ for all $i < j$.

What it says in plain English: For each edge in the graph, we have an ordered list of estimation procedures. The first procedure is fast but gives loose bounds. The second is slower but gives tighter bounds. And so on. Each subsequent estimator's bounds are guaranteed to be within (or equal to) the previous estimator's bounds.

Symbol-by-symbol breakdown:

- Θ: the function that maps edges to their sequences of estimators.
- $e \in E$: an edge in the graph.
- θ_e^i: the i-th estimator for edge e. This is a *procedure* that can be called to obtain bounds on $c(e)$.
- $k(e) \in \mathbb{N}$: the number of estimators available for edge e. Different edges may have different numbers of estimators.
- (l_e^i, u_e^i): the lower bound and upper bound returned by the i-th estimator.
- $0 \leq l_e^i \leq c(e) \leq u_e^i$: the bounds are *valid*—the true cost is guaranteed to lie within the interval $[l_e^i, u_e^i]$.
- $[l_e^j, u_e^j] \subseteq [l_e^i, u_e^i]$ for $i < j$: later estimators produce *tighter* (or equally tight) intervals. The lower bound can only increase, and the upper bound can only decrease.

Estimators for a Single Edge

Consider an edge e with true cost $c(e) = 7$, and three estimators:

Estimator	Runtime	Lower bound l_e^i	Upper bound u_e^i	Interval
θ_e^1	1 ms	3	12	$[3, 12]$
θ_e^2	10 ms	5	9	$[5, 9]$
θ_e^3	100 ms	6	8	$[6, 8]$

Notice that:

- The true cost $c(e) = 7$ lies within every interval: $[3, 12]$, $[5, 9]$, and $[6, 8]$.
- Each subsequent interval is contained within the previous: $[6, 8] \subset [5, 9] \subset [3, 12]$.
- The runtimes increase: 1 ms, 10 ms, 100 ms.

Estimators Do Not Necessarily Reveal the True Cost

Even after applying *all* available estimators, the true cost $c(e)$ may not be known exactly. The tightest interval $[l_e^{k(e)}, u_e^{k(e)}]$ may still have $l_e^{k(e)} < u_e^{k(e)}$. In the example above, even the most expensive estimator only narrows the cost to $[6, 8]$, not the exact value 7. The standard shortest path problem is a special case where every edge has a single estimator with $l_e^1 = u_e^1 = c(e)$.

2.2 Estimated Weighted Digraphs (Definition 2)

Definition 2.2 (Estimated Weighted Digraph)**.** An **estimated weighted digraph** is a tuple $G = (V, E, \Theta)$, where V and E are sets of vertices and edges (respectively), and Θ is a cost estimators function for E.

What this means: This is simply the generalization of a weighted graph where instead of having exact edge costs, we have estimator functions. Everything else about the graph (vertices, edges, directed structure) remains the same.

2.3 Tightest Bounds Over a Subset of Estimators (Definitions 3–4)

Once we have applied some (but not necessarily all) estimators for an edge, we want to extract the best (tightest) bounds available.

Definition 2.3 (Tightest Edge Bounds)**.** Let $\Phi(e)$ be a non-empty subset of estimators from $\Theta(e)$ for an edge e. The **tightest lower bound** and **tightest upper bound** over all estimators in $\Phi(e)$ are:

$$l_{\Phi(e)} := \max\{l_e^i \mid \theta_e^i = (l_e^i, u_e^i) \in \Phi(e)\}$$
$$u_{\Phi(e)} := \min\{u_e^j \mid \theta_e^j = (l_e^j, u_e^j) \in \Phi(e)\} \quad \text{(Eq. 4)}$$

What it computes: Given a set of estimators that have been applied to an edge, the tightest lower bound is the *maximum* of all the individual lower bounds, and the tightest upper bound is the *minimum* of all the individual upper bounds.

Why maximum for lower bounds and minimum for upper bounds?

- Each lower bound l_e^i satisfies $l_e^i \leq c(e)$. Since they are all valid lower bounds, the *largest* one is the tightest—it gets closest to $c(e)$ from below without exceeding it.
- Each upper bound u_e^j satisfies $u_e^j \geq c(e)$. The *smallest* one is the tightest—it gets closest to $c(e)$ from above without going below it.

Numerical Example

Suppose we have applied two estimators for an edge:

- θ_e^1 returned $(l_e^1, u_e^1) = (3, 12)$.
- θ_e^2 returned $(l_e^2, u_e^2) = (5, 9)$.

Then:

$$l_{\Phi(e)} = \max\{3, 5\} = 5 \quad \text{(tightest lower bound)}$$
$$u_{\Phi(e)} = \min\{12, 9\} = 9 \quad \text{(tightest upper bound)}$$

Our best knowledge of the edge cost is: $c(e) \in [5, 9]$.

Note: because estimator intervals are monotonically tightening, in this case θ_e^2 alone already provides the tightest bounds. But in more complex scenarios (e.g., if estimators are applied in non-sequential order), the max/min formulation handles everything correctly.

Path bounds (Equation 5): The bounds extend naturally to paths. For a path $p = \langle e_1, \ldots, e_n \rangle$, let $\Phi(p) = \bigcup_{e \in p} \Phi(e)$ be the union of all estimators applied to edges in p. The path lower and upper bounds are simply the sums of the individual edge bounds:

$$l_{\Phi(p)} := \sum_{i=1}^{n} l_{\Phi(e_i)}, \quad u_{\Phi(p)} := \sum_{i=1}^{n} u_{\Phi(e_i)} \qquad \text{(Eq. 5)}$$

What this says: The cheapest the path could possibly be is the sum of all the tightest lower bounds on its edges. The most expensive it could possibly be is the sum of all the tightest upper bounds.

Numerical Example

A path has three edges with tightest bounds $[3, 7]$, $[2, 5]$, and $[4, 6]$:

$$l_{\Phi(p)} = 3 + 2 + 4 = 9$$
$$u_{\Phi(p)} = 7 + 5 + 6 = 18$$

The true path cost is somewhere in $[9, 18]$.

The notation $\Phi^*(p)$ denotes the maximal-size $\Phi(p)$, which includes *all* estimators for all edges in p. This gives the tightest possible bounds for the path.

2.4 Connection to Standard Shortest Paths

Estimated weighted digraphs are a strict generalization of standard weighted digraphs.

Special Case: Standard Graphs

A standard weighted digraph is the special case where every edge has exactly one estimator ($k(e) = 1$ for all e) whose lower and upper bounds are both equal to the true cost: $\theta_e^1 = (c(e), c(e))$, i.e., $l_e^1 = u_e^1 = c(e)$.

In this case, the lower bound and upper bound of any path are both equal to its true cost, so the standard shortest path problem is recovered exactly.

3 The SLB Problem: Finding the Tightest Lower Bound

3.1 Why Lower Bounds on Optimal Cost Matter

In estimated weighted digraphs, we generally cannot determine the exact optimal cost C^* because edge costs are not known precisely. However, we can compute *bounds* on C^*.

Why is a lower bound on C^* useful?

- It provides a **quality guarantee** for any solution found: if we find a path with upper bound u and know that $C^* \geq L$, then the path is at most u/L times worse than optimal.
- It is the foundation for **bounded-suboptimality search**: algorithms that guarantee solutions within a factor $\mathcal{B}$ of optimal.
- Even when exact costs are unknown, the tightest lower bound gives us the best possible guarantee on what the optimal cost could be.

3.2 The Shortest Path Tightest Lower Bound (SLB) Problem (Problem 1)

Problem 1 (Shortest Path Tightest Lower Bound — SLB)**.** *Let $P = (G, v_s, V_g)$ where $G = (V, E, \Theta)$ is an estimated weighted digraph with cost estimators functions Θ, $v_s \in V$ is the start vertex, and $V_g \subset V$ is a set of goal vertices.*

*The **SLB problem** is to find a path π from v_s to some $v \in V_g$ such that π has the* tightest *lower bound of any path from v_s to $v \in V_g$, with respect to Θ. That is, $l(\pi) = L^*$ with:*

$$L^* := \min_{\pi'} \{ l_{\Phi^*(\pi')} \mid \pi' \text{ is a path from } v_s \text{ to } v \in V_g \} \qquad \text{(Eq. 6)}$$

What this says in plain English: Among all paths from the start to any goal, find the one whose tightest possible lower bound (computed using all available estimators) is the smallest. This path and its lower bound give us L^*.

Symbol-by-symbol breakdown:

- π': a candidate path from v_s to some goal vertex $v \in V_g$.
- $l_{\Phi^*(\pi')}$: the tightest lower bound of π' when *all* estimators for all edges in π' have been applied (the maximal Φ).
- $\min_{\pi'}$: take the minimum over all possible paths from start to goal.
- L^*: the *tightest lower bound on the optimal cost* C^*.

Why Min and Not Max?

The use of min here may seem counter-intuitive. If we want the "tightest" lower bound, should we not *maximize* it?
The key insight: L^* is the tightest lower bound on the *optimal cost* C^*. The optimal path π^* has some lower bound $l_{\Phi^*(\pi^*)}$. Ideally we would report this value, but π^* itself is unknown! Instead, we must consider *all* paths to goals. The path with the *smallest* tightest lower bound gives the tightest guarantee that applies to C^*, because $C^* = c(\pi^*) \geq l_{\Phi^*(\pi^*)} \geq L^*$.
In other words: L^* is a lower bound on the cost of *every* path (since the shortest path's lower bound cannot exceed the shortest path's true cost), and therefore it lower-bounds C^* itself.

3.3 SLB Generalizes Standard Shortest Paths (Theorem 1)

Theorem 3.1. *Problem 1 (SLB) generalizes GDS^3P problems.*

What this means: Every standard shortest path problem can be expressed as an SLB problem (but not vice versa). Therefore, SLB is at least as hard as the standard shortest path problem.

Proof sketch: In the standard case, each edge has one estimator with $l_e^1 = u_e^1 = c(e)$. The SLB solution is the path minimizing $\sum l_e^1 = \sum c(e) = c(\pi)$, which is exactly the shortest path. So a standard GDS^3P instance is a special case of SLB.

3.4 $\mathcal{B}$-Admissible Shortest Paths (Equation 7)

Once we have a lower bound L^*, we can define the notion of a bounded-suboptimal solution.

A path π is called a **$\mathcal{B}$-admissible shortest path** if its cost is bounded by a factor $\mathcal{B}$ of the optimal cost:

$$c(\pi) \leq C^* \times \mathcal{B} \tag{Eq. 7}$$

Symbol-by-symbol breakdown:

- $c(\pi)$: the true cost of path π.
- C^*: the optimal (minimum) cost of any path from start to goal.
- $\mathcal{B} \geq 1$: the suboptimality factor. $\mathcal{B} = 1$ means the path must be optimal. $\mathcal{B} = 2$ means the path can be at most twice the optimal cost.

Since we generally cannot compute $c(\pi)$ exactly in estimated weighted digraphs, we instead use upper bounds. To prove that π is $\mathcal{B}$-admissible, it suffices to show (Equation 8 in the paper):

$$u_{\Phi^*(\pi)} \leq L^* \times \mathcal{B} \tag{Eq. 8}$$

Why this works: Since $c(\pi) \leq u_{\Phi^*(\pi)}$ (the upper bound is valid) and $L^* \leq C^*$ (the lower bound is valid), we have:

$$c(\pi) \leq u_{\Phi^*(\pi)} \leq L^* \times \mathcal{B} \leq C^* \times \mathcal{B}$$

So the path is indeed $\mathcal{B}$-admissible.

Using the SLB Solution for Admissibility

Suppose SLB returns $L^* = 7$ and we find a path π with upper bound $u_{\Phi^*(\pi)} = 13$. The admissibility factor is:

$$\mathcal{B}(\pi) = \frac{u_{\Phi^*(\pi)}}{L^*} = \frac{13}{7} \approx 1.86$$

This means π is guaranteed to be at most 1.86 times the optimal cost. If we require $\mathcal{B} \leq 1.5$, this path does not qualify—we need a better path or tighter estimates.

3.5 A Complete SLB Example (Example 1 from the Paper)

Full SLB Example from the Paper

Consider an estimated weighted digraph $G = (V, E, \Theta)$ with:

- $V = \{v_0, v_1, v_2, v_3, v_4\}$
- $E = \{e_{01}, e_{02}, e_{14}, e_{21}, e_{23}, e_{24}\}$

The true (unknown) edge costs and estimators are:

	e_{01}	e_{02}	e_{14}	e_{21}	e_{23}	e_{24}
True cost c	4	4	5	3	7	6
θ^1 bounds	$(4, 4)$	$(2, 6)$	$(1, 10)$	$(2, 3)$	$(5, 9)$	$(4, 6)$
θ^2 bounds	—	$(3, 5)$	$(4, 6)$	$(3, 3)$	$(7, 8)$	—

Start: $v_s = v_0$. Goals: $V_g = \{v_3, v_4\}$.

Paths to goals and their tightest lower bounds (using all estimators):

1. Path $\pi_1 = \langle e_{02}, e_{24} \rangle$ ($v_0 \to v_2 \to v_4$):
   - Edge e_{02}: tightest lower bound $= \max(2, 3) = 3$ (from θ^1 and θ^2).
   - Edge e_{24}: tightest lower bound $= 4$ (only one estimator).
   - Path lower bound: $l_{\Phi^*(\pi_1)} = 3 + 4 = 7$.
2. Path $\pi_2 = \langle e_{02}, e_{23} \rangle$ ($v_0 \to v_2 \to v_3$):
   - Edge e_{02}: tightest lower bound $= 3$.
   - Edge e_{23}: tightest lower bound $= \max(5, 7) = 7$.
   - Path lower bound: $l_{\Phi^*(\pi_2)} = 3 + 7 = 10$.
3. Path $\pi_3 = \langle e_{01}, e_{14} \rangle$ ($v_0 \to v_1 \to v_4$):
   - Edge e_{01}: tightest lower bound $= 4$.
   - Edge e_{14}: tightest lower bound $= \max(1, 4) = 4$.
   - Path lower bound: $l_{\Phi^*(\pi_3)} = 4 + 4 = 8$.
4. Other paths (via more hops) would have higher lower bounds.

The minimum tightest lower bound is $L^* = \min\{7, 10, 8, \ldots\} = 7$, achieved by path $\pi_1 = \langle e_{02}, e_{24} \rangle$.

Interpreting the result:

- We know $C^* \geq L^* = 7$.
- The true optimal cost is $C^* = c(\pi^*) = c(\langle e_{01}, e_{14} \rangle) = 4 + 5 = 9$.
- But we could not determine this from estimators alone!
- For π_2: upper bound $u_{\Phi^*(\pi_2)} = \min(5, 5) + \min(8, 8) = 5 + 8 = 13$, so $\mathcal{B}(\pi_2) = 13/7 \approx 1.86$.

4 Algorithm 1: BEAUTY

BEAUTY stands for **B**ranch-and-bound **E**stimation **A**pplied in **U**CS **T**o **Y**ield bottom—a variant of Uniform-Cost Search (UCS) that dynamically applies cost estimators during search to solve the SLB problem.

4.1 Review: How UCS Works

Before explaining BEAUTY, let us briefly review Uniform-Cost Search, since BEAUTY extends it.

Uniform-Cost Search (Dijkstra's Algorithm)

UCS maintains two data structures:

- **OPEN**: a priority queue of nodes to be explored, ordered by their accumulated path cost $g(n)$ (lowest cost first).
- **CLOSED**: a set of nodes that have already been expanded.

The algorithm repeats: pop the node n with the smallest $g(n)$ from OPEN, check if it is a goal, and if not, expand it by generating its successors and inserting them into OPEN with their accumulated costs. When a goal node is popped, the optimal path has been found.

In standard UCS, when we expand node n and generate successor s via edge $e = (n, s)$, we compute $g(s) = g(n) + c(e)$ using the exact edge cost.

4.2 BEAUTY: The Key Differences from UCS

BEAUTY modifies UCS in two critical ways:

1. **Incremental estimation instead of exact cost:** When generating successor s of node n via edge e, BEAUTY does not compute $c(e)$ exactly. Instead, it iterates through the estimators $\theta_e^1, \theta_e^2, \ldots$ one at a time, obtaining increasingly tight lower bounds on $c(e)$. It uses these lower bounds to compute a lower bound on $g(s)$, namely $\tilde{g}_l = g_l(n) + l(e)$, where $g_l(n)$ is the accumulated lower bound to node n and $l(e)$ is the current tightest lower bound for edge e.
2. **Early termination of estimation:** If, during estimation, the accumulated lower bound $\tilde{g}_l$ already exceeds the best known path cost to s (i.e., $\tilde{g}_l \geq g_l(s)$), then there is no need to apply further (expensive) estimators—this path to s via n is already known to be no better than a previously found path. The while loop exits early.

4.3 BEAUTY Pseudocode Walkthrough

Let us walk through the algorithm step by step, using the pseudocode from the paper.

Initialization (Lines 1–2):

- $g_l(s_0) \leftarrow 0$: the lower bound on the cost to reach the start node is 0.
- OPEN $\leftarrow \emptyset$; CLOSED $\leftarrow \emptyset$: both lists start empty.
- Insert s_0 into OPEN with key $g_l(s_0) = 0$.

Main loop (Lines 3–23): While OPEN is not empty:

1. **Line 4:** Pop node n with the minimal $g_l(n)$ from OPEN (best-first order by lower bound).
2. **Lines 5–8 (Goal test):** If n is a goal node, we have found a candidate solution. Set $l(\pi) \leftarrow g_l(n)$ (the path lower bound). Then call BEAUTY-PS (the post-search procedure) to tighten the bounds and determine optimality. Return the result.
3. **Line 9:** Insert n into CLOSED (it has been expanded).
4. **Lines 10–23 (Successor generation):** For each successor s of n:
   - **Lines 11–12:** If s is new (not in OPEN or CLOSED), initialize $g_l(s) \leftarrow \infty$.
   - **Line 13:** Initialize $\tilde{g}_l \leftarrow g_l(n)$ (the accumulated lower bound to the parent node n).
   - **Lines 14–18 (Estimation loop):** While $\tilde{g}_l < g_l(s)$ and estimators remain for edge $e = (n, s)$:
     - **Line 15:** $l(e) \leftarrow$ Apply the next estimator for e to get a (tighter) lower bound.
     - **Line 16:** Update $\tilde{g}_l \leftarrow g_l(n) + l(e)$.
     - **Key insight:** If $\tilde{g}_l > l_{est}$, **break** out of the loop (Line 17–18). This is the enhanced setting threshold.
   - **Lines 19–23 (Duplicate detection):** If $\tilde{g}_l < g_l(s)$ *and* $\tilde{g}_l \leq l_{prune}$ (this path to s is better than the previous best and within the pruning threshold), update $g_l(s) \leftarrow \tilde{g}_l$ and insert/update s in OPEN. Otherwise, this path is no better or exceeds the pruning bound—discard it.

BEAUTY's Core Savings

The main computational saving comes from the estimation loop (Lines 14–18). In standard UCS (or the baseline EI-UCS), every edge gets fully estimated using all estimators. BEAUTY stops applying estimators as soon as the current lower bound already rules out this path as an improvement. This means expensive estimators are *never applied to edges that do not matter*.

4.4 The Enhanced Setting: l_{est} and l_{prune}

BEAUTY has two hyper-parameters, l_{est} and l_{prune}, that provide additional control:

- l_{est}: an upper bound on the lower-bound accumulation for activating estimators. When a node n has $g_l(n) > l_{est}$, we do not apply further estimators when generating its successors—we only use the first (cheapest) estimator. **Intuition:** if the accumulated lower bound is already large, this node is unlikely to be on the optimal path, so spending time on tight estimates is wasteful.
- l_{prune}: a pruning threshold. If $\tilde{g}_l > l_{prune}$, the successor is not added to OPEN at all. **Intuition:** if the lower bound already exceeds l_{prune}, this path cannot lead to a solution better than the threshold, so we prune it.

Base setting: $l_{est} = l_{prune} = \infty$ (no thresholds, all estimators applied, no pruning). In this setting, BEAUTY is essentially UCS with incremental estimation.

Enhanced setting: Setting l_{est} and l_{prune} to finite values reduces computation but requires knowledge of (or an approximation to) L^*. Setting $l_{est} \geq L^*$ ensures optimality (Lemma 3); setting $l_{prune} \geq L^*$ ensures completeness (Lemma 1).

4.5 BEAUTY-PS: The Post-Search Procedure

When BEAUTY finds a goal node, it calls BEAUTY-PS (Procedure 2 in the paper) to tighten the lower bound of the solution path.

What BEAUTY-PS does:

1. Set $Opt \leftarrow true$ and $\underline{l}^* \leftarrow l(\pi)$ (the current lower bound).
2. For each edge e in the solution path π:
   (a) If unused estimators remain, apply the best (most expensive unused) estimator.
   (b) Update $l(\pi)$ with the new lower bound.
3. If $l(\pi) > \underline{l}^*$ (the lower bound increased), set $Opt \leftarrow false$ and update $\bar{l}^* \leftarrow l(\pi)$.
4. Return Opt, $\underline{l}^*$, and $\bar{l}^*$.

Why this matters: After the search phase, the solution path may have edges where only cheap estimators were used. BEAUTY-PS applies all remaining estimators to the solution path's edges to get the tightest possible lower bound. If this tightened bound equals the original $\underline{l}^*$, the solution is provably optimal. If it increases, the solution *might* not be optimal—the tightened bound $\bar{l}^*$ serves as an upper bound on L^*.

BEAUTY Trace (Example 2 from the Paper)

Running BEAUTY on the graph from Example 1 (Section 3.5) with $l_{est} = l_{prune} = \infty$ (base setting):
Iteration 1: Pop v_0 from OPEN (key 0). Invoke estimators $\theta^1_{e_{01}}$, $\theta^1_{e_{02}}$, and $\theta^2_{e_{02}}$. Insert v_1, v_2 into OPEN with keys $4, 3$.

- Edge e_{01}: apply $\theta^1_{e_{01}}$, get $l = 4$. Only one estimator, so done. $g_l(v_1) = 0 + 4 = 4$.
- Edge e_{02}: apply $\theta^1_{e_{02}}$, get $l = 2$. Since $\tilde{g}_l = 2 < \infty = g_l(v_2)$, apply $\theta^2_{e_{02}}$, get $l = 3$. $g_l(v_2) = 0 + 3 = 3$.

Iteration 2: v_2 is removed from OPEN (key 3). Invoke estimators $\theta^1_{e_{21}}$, $\theta^1_{e_{23}}$, $\theta^2_{e_{23}}$, $\theta^1_{e_{24}}$. Insert v_3, v_4 into OPEN with keys $10, 7$.

- Edge e_{21}: apply $\theta^1_{e_{21}}$, get $l = 2$. $\tilde{g}_l = 3 + 2 = 5 > 4 = g_l(v_1)$. Since $\tilde{g}_l \geq g_l(v_1)$, the while loop exits immediately and v_1 is not updated.
- Edge e_{23}: apply $\theta^1_{e_{23}}$, get $l = 5$. $\tilde{g}_l = 3 + 5 = 8 < \infty = g_l(v_3)$, apply $\theta^2_{e_{23}}$, get $l = 7$. $g_l(v_3) = 3 + 7 = 10$.
- Edge e_{24}: apply $\theta^1_{e_{24}}$, get $l = 4$. $g_l(v_4) = 3 + 4 = 7$.

Iteration 3: v_1 is removed from OPEN (key 4). Invoke estimators $\theta^1_{e_{14}}$ and $\theta^2_{e_{14}}$.

- Edge e_{14}: apply $\theta^1_{e_{14}}$, get $l = 1$. $\tilde{g}_l = 4 + 1 = 5 < 7 = g_l(v_4)$. Apply $\theta^2_{e_{14}}$, get $l = 4$. $\tilde{g}_l = 4 + 4 = 8 > 7 = g_l(v_4)$. So this path is not better.

Iteration 4: v_4 is removed from OPEN (key 7). v_4 is a goal! $l(\pi) = 7$. Call BEAUTY-PS on path $\pi = \langle e_{02}, e_{24} \rangle$. Both edges are already fully estimated, so $l(\pi) = 7 = \underline{l}^*$. $Opt = true$; BEAUTY returns $\langle e_{02}, e_{24} \rangle$, *true*, 7, 7.

4.6 Theoretical Properties of BEAUTY

The paper proves three properties about BEAUTY:

Lemma 4.1 (Conditional Completeness — Lemma 1)**.** *BEAUTY, provided with $l_{prune} \geq L^*$, is **complete**: if a solution exists, BEAUTY will find one.*

Plain English: As long as the pruning threshold is not set below the optimal lower bound, BEAUTY will not accidentally prune the optimal path. Since BEAUTY explores nodes in best-first order (by lower bound), and no valid path is pruned, a goal node will eventually be reached.

Lemma 4.2 (Bounds for L^* — Lemma 2)**.** *BEAUTY, provided with $l_{prune} \geq L^*$, returns bounds $0 \leq \underline{l}^* \leq L^* \leq \bar{l}^*$, if a solution exists. Furthermore, if $l_{est} < L^*$ also holds, then $\underline{l}^* > l_{est}$.*

Plain English: BEAUTY returns a lower bound $\underline{l}^*$ and an upper bound $\bar{l}^*$ that bracket L^*. The lower bound comes from the search (best-first order ensures $g_l(n) \leq L^*$ when n is popped), and the upper bound comes from BEAUTY-PS (tightening the solution path's bounds).

Lemma 4.3 (Conditional Optimality — Lemma 3)**.** *BEAUTY, provided with $l_{prune} \geq L^*$ and $l_{est} \geq L^*$, returns the optimal solution π and $\bar{l}^* = L^*$, if a solution exists.*

Plain English: If both thresholds are at least L^*, then BEAUTY finds the exact SLB solution. The condition $l_{est} \geq L^*$ ensures that all relevant estimators are applied (none are skipped due to the threshold), and $l_{prune} \geq L^*$ ensures no valid paths are pruned.

Setting the Thresholds to ∞

In the base setting ($l_{est} = l_{prune} = \infty$), both conditions in Lemma 3 are trivially satisfied. So BEAUTY in its base setting is always complete and optimal. The tradeoff: with ∞ thresholds, BEAUTY applies more estimators and explores more nodes than necessary, losing potential speedups.

5 Algorithm 3: A-BEAUTY (Anytime BEAUTY)

5.1 The Problem with Setting Thresholds

BEAUTY's enhanced setting requires $l_{est} \geq L^*$ and $l_{prune} \geq L^*$ for optimality. But L^* is unknown—it is exactly what we are trying to compute!

A-BEAUTY solves this chicken-and-egg problem using an **anytime** approach: run BEAUTY iteratively, using the results of each iteration to set better thresholds for the next.

5.2 A-BEAUTY Algorithm Walkthrough

A-BEAUTY (Algorithm 3 in the paper) works as follows:

Initialization:

- $\underline{l}^* \leftarrow 0$: initial lower bound on L^*.
- $\bar{l}^* \leftarrow \infty$: initial upper bound on L^*.
- $Opt \leftarrow false$: optimality not yet proven.

Main loop (while not Opt):

1. Call BEAUTY with current bounds: $l_{est} = \underline{l}^*$ and $l_{prune} = \bar{l}^*$.
2. BEAUTY returns a path π, Opt, $\underline{l}^*$, and a new $\bar{l}$.
3. If $\pi = \emptyset$ (no solution found), return failure.
4. If the new upper bound $\bar{l}$ is less than the current $\bar{l}^*$, update $\bar{l}^* \leftarrow \bar{l}$.
5. Print current solution π, $\underline{l}^*$, $\bar{l}^*$ (anytime output).

Key properties of the loop:

- $\underline{l}^*$ is **monotonically non-decreasing**: each iteration can only maintain or increase the lower bound, because BEAUTY-PS tightens bounds.
- $\bar{l}^*$ is **monotonically non-increasing**: each iteration can only maintain or decrease the upper bound.
- The gap $\bar{l}^* - \underline{l}^*$ shrinks with each iteration.
- Convergence: the bounds come from a finite set of values (determined by Θ), so the process must terminate in finitely many iterations.

A-BEAUTY Iterations (Example 3 from the Paper)

Running A-BEAUTY on the graph from Example 1:
Iteration 1: Call BEAUTY with $l_{est} = 0$, $l_{prune} = \infty$.
- Since $l_{est} = 0$, only the first (cheapest) estimator is applied per edge during the search phase (the l_{est} threshold immediately stops further estimation).
- The utilized estimators are: $\theta^1_{e_{01}}, \theta^1_{e_{02}}, \theta^1_{e_{14}}, \theta^1_{e_{21}}, \theta^1_{e_{23}}$, and $\theta^1_{e_{24}}$, where $\theta^2_{e_{14}}$ is additionally invoked by BEAUTY-PS.
- The algorithm prints $\langle e_{01}, e_{14} \rangle$, $\underline{l}^* = 5$, $\bar{l}^* = 8$.

Iteration 2: Call BEAUTY with $l_{est} = 5$, $l_{prune} = 8$.
- The higher l_{est} allows more estimators to be applied.
- The estimator $\theta^2_{e_{02}}$ is also utilized now (tightening e_{02}'s lower bound from 2 to 3).
- The algorithm prints $\langle e_{02}, e_{24} \rangle$, $\underline{l}^* = 7$, $\bar{l}^* = 7$ and returns $\langle e_{02}, e_{24} \rangle$, 7, 7.
- Since $\underline{l}^* = \bar{l}^*$, $Opt = true$. A-BEAUTY returns the optimal solution.

The final answer is: $L^* = 7$, achieved by path $v_0 \to v_2 \to v_4$.

5.3 A-BEAUTY: Theoretical Guarantees (Theorem 2)

Theorem 5.1 (Completeness, Soundness, and Optimality)**.** *A-BEAUTY is* ***complete****. If a solution exists for P, then a shortest path tightest lower bound π and L^* are returned.*

What this says: A-BEAUTY is guaranteed to find the optimal SLB solution, with no conditions on hyper-parameters (unlike BEAUTY, which needs $l_{est} \geq L^*$ and $l_{prune} \geq L^*$).

Why it works:

- In the first iteration, $l_{est} = 0$ and $l_{prune} = \infty$. By Lemma 1, BEAUTY is complete (since $l_{prune} = \infty \geq L^*$). By Lemma 2, it returns $\underline{l}^* > 0$ and $\bar{l}^* < \infty$.
- In each subsequent iteration, $l_{est} = \underline{l}^*$ and $l_{prune} = \bar{l}^*$. Since $\underline{l}^* \leq L^*$ and $\bar{l}^* \geq L^*$ (from Lemma 2), the conditions of Lemmas 1 and 2 remain satisfied.
- $\underline{l}^*$ monotonically increases, and since it takes values from a finite set (determined by the estimators), it must reach L^* in finitely many steps.
- When $\underline{l}^* = \bar{l}^*$, Lemma 3 implies optimality.

6 EI-UCS: The Baseline Algorithm

To understand the savings provided by BEAUTY and A-BEAUTY, it helps to understand the baseline they are compared against.

Estimation-time Indifferent UCS (EI-UCS)

EI-UCS is a straightforward adaptation of UCS to estimated weighted digraphs. For every edge encountered during the search, EI-UCS applies *all* available estimators to obtain the tightest possible lower bound. It then uses these tightest bounds in the standard UCS priority queue.
EI-UCS is correct and complete, but it is wasteful: it applies expensive estimators to many edges that turn out to be irrelevant (not on any near-optimal path).

The key difference between EI-UCS and BEAUTY is the condition $\tilde{g}_l < g_l(s)$ (Line 14 in Algorithm 1). In EI-UCS, this check is absent—every edge is fully estimated regardless of whether it could improve the path to its endpoint.

7 Empirical Results: Understanding the Savings

7.1 Experimental Setup

The experiments use AI planning benchmark problems from the International Planning Competition (IPC). Each edge in the planning graph has a single cost $c_{old}(e)$, which is transformed into three estimators as follows:

- The true cost becomes $c_{new}(e) \geq c_{old}(e) \times f_3$, with $f_3 > f_2 > f_1 \geq 1$.
- Estimator 1 (cheapest): lower bound $l_e^1 = c_{old} \times f_1$ (coarsest).
- Estimator 2: lower bound $l_e^2 = c_{old} \times f_2$.
- Estimator 3 (most expensive): lower bound $l_e^3 = c_{old} \times f_3$ (tightest).

This simulates a scenario where cheap estimators give loose bounds (low multipliers) and expensive estimators give tight bounds (high multipliers).

To diversify the estimator configurations across different edges, the parameters f_1, f_2, f_3 are chosen according to the result of a simple hash function:

$$\text{Hash} = (c_{old}(e) + \text{seed}) \mod 9 \tag{Eq. 9}$$

The configuration of f_1, f_2, f_3 is then set according to the hash value, with $f_1 \in \{1, 2, 3\}$, $f_2 \in \{f_1+1, f_1+2, f_1+3\}$, and $f_3 \in \{f_2+1\}$, resulting in nine different configurations. Each problem was run once per seed, where the seeds were taken from the set $[0, 8]$, yielding 9 instances per problem.

The primary metric is r_{L_3}, the ratio of expensive (third-layer) estimator usage:

$$r_{L_3}(\text{Alg}) := \frac{L_3(\text{Alg})}{L_3(\text{EI-UCS})}$$

where $L_3(\text{Alg})$ is the number of third-layer estimators applied by algorithm Alg.

7.2 Key Results

Algorithm	**Avg. r_{L_3} (%)**	**Interpretation**
EI-UCS	100%	Baseline: applies all estimators everywhere
BEAUTY ($l_{est} = l_{prune} = \infty$)	60.82%	Saves ∼39% of expensive estimations
A-BEAUTY-2 (2 iterations)	46.03%	Saves ∼54% of expensive estimations
A-BEAUTY-10 (10 iterations)	45.13%	Saves ∼55% of expensive estimations

Interpretation:

- BEAUTY in its base setting already avoids about 40% of expensive estimator calls, simply by stopping estimation when a path is provably non-improving.
- A-BEAUTY-2 provides an additional 15% savings on top of BEAUTY, by using the first iteration's bounds to set better thresholds.
- A-BEAUTY-10 provides only marginal improvement over A-BEAUTY-2, suggesting that most of the benefit is captured in the first two iterations.
- The node expansion cost increases (roughly 1.8× for A-BEAUTY-2 and 8.5× for A-BEAUTY-10 compared to EI-UCS) because the search is restarted each iteration. But since the edge estimation savings dominate ($\tau_w \gg \tau_v$ in practice), the overall runtime still decreases.

The Tradeoff: Search vs. Estimation

A-BEAUTY reduces estimation time (T_w) but increases search time (T_v) because each iteration restarts the search. This is beneficial when $\tau_w \gg \tau_v$ (edge estimation is much more expensive than search operations). In domains where edge evaluation is cheap, the overhead of restarting may not be worthwhile.

7.3 BEAUTY-PS Alone Is Remarkably Effective

An interesting finding from the experiments:

One Iteration of BEAUTY + BEAUTY-PS ≈ Near-Optimal

When BEAUTY is run once with $l_{est} = 0$ and $l_{prune} = \infty$ (the least informed settings), followed by BEAUTY-PS on the solution path, the returned $\bar{l}^*$ is on average only $1.0082 \times L^*$—less than 1% above the true optimal lower bound.
This means that a single, cheap pass of BEAUTY followed by tightening only the solution path produces a very good approximation of L^*, at minimal estimation cost.

8 How BEAUTY Differs from Standard UCS: A Side-by-Side Comparison

Aspect	**Standard UCS**	**BEAUTY**
Edge cost	Single exact value $c(e)$	Multiple estimators $\theta_e^1, \ldots, \theta_e^{k(e)}$
Cost used for ordering	$g(n) = \sum c(e_i)$ (exact)	$g_l(n) = \sum l_{\Phi(e_i)}$ (lower bound)
When generating successor s	Compute $c(e)$ once, set $g(s)$	Iterate through estimators, stop when $\tilde{g}_l \geq g_l(s)$
Duplicate detection	Compare $g(s)$ values	Compare $\tilde{g}_l$ with $g_l(s)$
Goal reached ⇒	Optimal path found	Call BEAUTY-PS to tighten bounds
Output	Optimal path and cost C^*	Path π, bounds $\underline{l}^* \leq L^* \leq \bar{l}^*$

9 Theoretical Connections and Complexity

9.1 SLB is at Least as Hard as Shortest Paths

Since every GDS^3P instance can be expressed as an SLB instance (Theorem 1), the complexity of SLB is at least that of the shortest path problem. This means $O((|V| + |E|) \log |V|)$ with a binary heap, or $O(|E| + |V| \log |V|)$ with a Fibonacci heap, is a lower bound on the time complexity (ignoring estimation costs).

In practice, the total cost is:

$$T = \underbrace{O((|V| + |E|) \log |V|)}_{\text{search operations}} + \underbrace{\sum_{e \in E'} \sum_{i=1}^{k_e} \tau_{w_i}(e)}_{\text{estimation operations}}$$

where $E' \subseteq E$ is the set of edges that are actually estimated, and $k_e \leq k(e)$ is the number of estimators applied to edge e. BEAUTY and A-BEAUTY aim to minimize the second term by reducing both $|E'|$ and k_e.

9.2 Why Branch-and-Bound?

The name BEAUTY refers to "Branch-and-bound Estimation." The connection to branch-and-bound is:

Branch-and-Bound in BEAUTY

In classical branch-and-bound:

- **Branching:** Divide the search space into subproblems (here: expanding a node creates successor subproblems).
- **Bounding:** Compute bounds on each subproblem's optimal value (here: compute lower bounds on path costs using estimators).
- **Pruning:** Discard subproblems whose bounds prove they cannot contain the optimal solution (here: stop applying estimators when $\tilde{g}_l \geq g_l(s)$, or prune when $\tilde{g}_l > l_{prune}$).

The "branch-and-bound" in BEAUTY is applied to the *estimation process* itself: instead of branching over solution paths, BEAUTY "branches" over estimators for each edge and "bounds" the path cost to decide whether further estimation is worthwhile.

10 Related Concepts and Extensions

10.1 Relationship to Lazy Search

The SLB framework is related to *lazy evaluation* approaches in search:

- **Lazy weighted A*** and similar algorithms delay edge evaluation until necessary. They assume edges are expensive to evaluate but have a single exact cost.
- **BEAUTY** generalizes this by allowing *multiple levels of laziness*: instead of a binary choice (evaluate or not), BEAUTY can evaluate at any of $k(e)$ accuracy levels.

10.2 Relationship to Fuzzy and Random Weights

The paper notes several other extensions to standard edge weights:

- **Random weights:** Each edge's cost is drawn from a distribution. The goal is typically to minimize the expected cost.
- **Fuzzy weights:** Edge costs are represented as fuzzy numbers (approximate ranges with membership functions).
- **Multi-objective weights:** Each edge carries a vector of costs, and the goal is to optimize multiple objectives simultaneously.

The key distinction of the SLB framework is that it focuses on the *computation time* of edge weights, not their uncertainty per se. The bounds come from the computational budget spent on estimation, not from inherent randomness or fuzziness in the costs.

11 Summary of Notation

For quick reference, here is a table of all major symbols used in the paper:

Symbol	Type / Domain	Meaning
(V, E, c)	Graph components	Vertices, edges, cost function (standard graph)
(V, E, Θ)	Graph components	Estimated weighted digraph
$c(e) \in \mathbb{R}^+$	Scalar	True (unknown) cost of edge e
$c(p) = \sum c(e_k)$	Scalar	True cost of path p
C^*	Scalar	Optimal path cost from v_s to V_g
π, π^*	Path (edge sequence)	A path; the optimal path
v_s	Vertex	Start vertex
$V_g \subset V$	Set of vertices	Goal vertices
Θ	Function $E \to$ sequences	Cost estimators function
$\Theta(e)$	Sequence of procedures	All estimators for edge e
θ_e^i	Procedure	The i-th estimator for edge e
$k(e) \in \mathbb{N}$	Integer	Number of estimators for edge e
(l_e^i, u_e^i)	Pair of scalars	Bounds from i-th estimator
$\Phi(e)$	Subset of $\Theta(e)$	Applied estimators for edge e
$\Phi^*(p)$	Maximal $\Phi(p)$	All estimators for all edges in p
$l_{\Phi(e)}$	Scalar	Tightest lower bound on $c(e)$ from $\Phi(e)$
$u_{\Phi(e)}$	Scalar	Tightest upper bound on $c(e)$ from $\Phi(e)$
$l_{\Phi(p)}, u_{\Phi(p)}$	Scalars	Path lower/upper bounds
L^*	Scalar	Optimal tightest lower bound (SLB solution)
$\underline{l}^*$	Scalar	Current lower bound on L^*
$\overline{l}^*$	Scalar	Current upper bound on L^*
$\mathcal{B}$	Scalar ≥ 1	Suboptimality factor
$g_l(n)$	Scalar	Accumulated lower bound to node n
$\tilde{g}_l$	Scalar	Tentative accumulated lower bound (during estimation)
l_{est}	Scalar	Estimation threshold (BEAUTY parameter)
l_{prune}	Scalar	Pruning threshold (BEAUTY parameter)
$l(e)$	Scalar	Current tightest lower bound for edge e
$l(\pi)$	Scalar	Current tightest lower bound for path π
Opt	Boolean	Whether the solution is proven optimal
T	Time	Total search runtime
T_w, T_v	Time	Edge-weight computation time; vertex operation time
τ_w, τ_v	Time per operation	Average time per weight computation; per vertex op
w, v	Integer	Number of weight computations; vertex operations
r_{L_3}	Ratio $\in [0, 1]$	Ratio of expensive estimator usage vs. EI-UCS
r_{exp}	Ratio ≥ 0	Ratio of expanded nodes vs. EI-UCS
$L_3(\text{Alg})$	Integer	Number of third-layer estimators used by Alg

12 Putting It All Together: The Big Picture

The Paper's Main Contributions in One Page

1. **A new graph model:** *Estimated weighted digraphs* generalize standard weighted graphs by replacing exact edge costs with ordered sequences of estimators, each providing increasingly tight bounds at increasing computational cost.
2. **A new problem:** The *Shortest path Tightest Lower Bound (SLB)* problem asks for the path with the tightest lower bound on the optimal cost. This is a proper generalization of the standard shortest path problem.
3. **Two algorithms:**
   - **BEAUTY**: A UCS variant that incrementally applies estimators during search, stopping early when further estimation cannot improve the current best path. In its base setting, it saves ∼40% of expensive estimator applications.
   - **A-BEAUTY**: An anytime wrapper around BEAUTY that iteratively tightens thresholds, achieving ∼55% savings while guaranteeing optimality.
4. **Theoretical guarantees:** Both algorithms are proven complete and correct. A-BEAUTY converges to the optimal SLB solution in finitely many iterations.
5. **Practical impact:** In AI planning benchmarks with synthesized multi-level estimators, the algorithms dramatically reduce the number of expensive edge evaluations while finding provably optimal (or near-optimal) solutions.

The core insight of the paper is elegant: **not all edges need to be evaluated precisely**. Just as a human planning a road trip would first check a rough map to eliminate obviously bad routes before looking up detailed traffic data for the remaining candidates, BEAUTY and A-BEAUTY use cheap estimators to rule out unpromising edges and save expensive evaluations for the edges that actually matter.

This approach is particularly valuable in domains where edge evaluation dominates search time—such as robotics (where edge costs require physics simulations), AI planning (where action costs require model evaluations), and route planning (where travel times require API queries).