**Informed Search Algorithms – Research-Based Tasks**

**🔰 Level 1: Understanding Core Concepts**

**Task 1: What is a Heuristic?**

**Objective:**  
Research and define the term *heuristic* in the context of AI and search algorithms.

* What does a heuristic function do?
* How does it differ from path cost?
* Why is it used in informed search?

**Deliverable:**  
A one-page written explanation with real-world examples.

**📑 What is a Heuristic?**

In the context of **Artificial Intelligence (AI)** and **search algorithms**, a *heuristic* is a technique or function that helps guide a search process towards a goal by providing an estimate of how close a given state is to the desired goal state. Unlike uninformed search strategies, which explore the search space blindly, heuristics use additional problem-specific knowledge to make more intelligent choices about which paths to follow.

**📌 What Does a Heuristic Function Do?**

A **heuristic function (commonly denoted as h(n) in AI)** estimates the cost from a given node n in the search space to the goal. It doesn't compute the exact cost but provides an educated guess based on available information, which helps prioritize which paths are more promising to explore.

**For example:**

* In a GPS navigation system, the straight-line (Euclidean) distance from your current location to the destination is a heuristic — it estimates how far you are from your goal, even if actual travel roads might be longer.

**📌 How Does It Differ from Path Cost?**

It’s important to distinguish between a **path cost** and a **heuristic function**:

| **Aspect** | **Path Cost (g(n))** | **Heuristic Function (h(n))** |
| --- | --- | --- |
| **Definition** | Actual cost incurred to reach a node n from the starting point | Estimated cost from node n to the goal |
| **Based On** | Known, exact values (e.g., distances traveled, resources used) | Problem-specific knowledge or assumptions |
| **Usage** | Tracks the cost of the path so far | Predicts future cost to goal |

In informed search algorithms like **A\***, both values are combined:

f(n)=g(n)+h(n)f(n) = g(n) + h(n)

where f(n) is the total estimated cost of the cheapest solution through node n.

**📌 Why Is It Used in Informed Search?**

A heuristic function is essential in **informed search algorithms** because:

* It **improves search efficiency** by reducing the number of explored nodes.
* It **guides the search towards the goal faster** by evaluating which paths are likely to lead to an optimal solution.
* It makes algorithms like **A\***, **Greedy Best-First Search**, and others practical for large or complex problems by providing a strategy to decide which path to explore next.

Without heuristics, an AI would rely on **uninformed search strategies** (like Breadth-First or Depth-First Search), which may waste time and resources by exploring irrelevant paths.

**📌 Real-World Examples**

* **Route Planning in Maps:**  
  GPS uses estimated straight-line distance (heuristic) to quickly suggest the shortest route.
* **Puzzle Solvers (e.g., 8-Puzzle):**  
  The number of misplaced tiles or the total Manhattan distance of tiles from their goal positions are common heuristic functions.
* **Game AI (e.g., Chess):**  
  A heuristic might assign scores to board positions based on material value, control of the center, king safety, etc., to estimate which move is best.

**Task 2: Admissible vs Inadmissible Heuristics**

**Objective:**

* Research what it means for a heuristic to be admissible.
* Find at least **two examples** of admissible heuristics and **one inadmissible** heuristic.
* Explain the consequences of using an inadmissible heuristic in A\*.

**Deliverable:**  
Table comparing admissible/inadmissible heuristics with examples.

**📑 Admissible vs Inadmissible Heuristics**

**📌 What is an Admissible Heuristic?**

A **heuristic is admissible** if it **never overestimates the true cost** of reaching the goal from any given node.  
In other words:

For all nodes n, h(n)≤h∗(n)\text{For all nodes } n,\ h(n) \leq h^\*(n)

Where:

* h(n)h(n) = heuristic estimate of the cost from n to goal
* h∗(n)h^\*(n) = true cost from n to goal

Admissible heuristics are always **optimistic** — they either underestimate or exactly estimate the true cost but never exceed it.

**📌 What is an Inadmissible Heuristic?**

A **heuristic is inadmissible** if it can **overestimate the true cost** of reaching the goal from a given node.  
This means:

For some node n, h(n)>h∗(n)\text{For some node } n,\ h(n) > h^\*(n)

Inadmissible heuristics can mislead search algorithms by overvaluing certain paths, causing them to overlook the optimal solution.

**📊 Table: Comparison of Admissible vs Inadmissible Heuristics**

| **Aspect** | **Admissible Heuristics** | **Inadmissible Heuristics** |
| --- | --- | --- |
| **Definition** | Never overestimates the true cost to reach the goal | May overestimate the true cost to reach the goal |
| **Guarantee of Optimality in A\*** | Yes | No |
| **Example 1** | In 8-puzzle: Number of misplaced tiles | In 8-puzzle: Double the number of misplaced tiles |
| **Example 2** | In pathfinding: Straight-line (Euclidean) distance to goal | In pathfinding: Straight-line distance × 1.5 |
| **Use Case** | Used when optimal solution is critical | May be used when faster, approximate solutions are acceptable |
| **Risk** | Might explore more nodes due to underestimation | Can miss the optimal solution due to overestimation |

**📌 Examples**

**✔️ Admissible Heuristics:**

1. **Number of misplaced tiles** (8-puzzle)  
   — Counts how many tiles are out of place.  
   (Cannot overestimate moves needed to solve the puzzle.)
2. **Straight-line (Euclidean) distance** (Pathfinding)  
   — The direct distance between the current point and goal.  
   (Always underestimates actual road or path travel.)

**❌ Inadmissible Heuristic:**

1. **Double the number of misplaced tiles** (8-puzzle)  
   — Multiplies the count of misplaced tiles by 2.  
   (May overestimate actual moves needed, violating admissibility.)

**📌 Consequences of Using an Inadmissible Heuristic in A\***

* **Loss of Optimality:**  
  A\* search relies on admissible heuristics to guarantee the optimal path. An inadmissible heuristic can cause A\* to choose a suboptimal path as it might overvalue certain nodes.
* **Faster, Less Accurate Results:**  
  While inadmissible heuristics may speed up the search by aggressively pruning paths, they do so at the risk of missing the best solution.

**Example:**  
If a heuristic overestimates, A\* might prematurely discard the optimal path, settling for a longer one it believes is cheaper.

**Task 3: Consistent (Monotonic) Heuristic**

**Objective:**

* Define a consistent heuristic.
* What is the mathematical condition for consistency?
* Find a situation where a heuristic is admissible but not consistent.

**Deliverable:**  
Summary of findings + example scenario.

**📑 Consistent (Monotonic) Heuristic**

**📌 What is a Consistent Heuristic?**

A **consistent heuristic** (also known as a **monotonic heuristic**) is one where the estimated cost from the current node to the goal is **always less than or equal to the estimated cost from a neighboring node plus the actual cost to reach that neighbor**.

In simpler terms:

* The estimated cost should **never decrease more than the actual cost of moving to a neighbor**.
* This ensures that the total estimated cost (f(n) = g(n) + h(n)) **never decreases along a path** — hence, it's called *monotonic*.

**📌 Mathematical Condition for Consistency**

A heuristic h(n) is consistent if for every node n and every successor n' of n:

h(n)≤c(n,n′)+h(n′)h(n) \leq c(n, n') + h(n')

Where:

* h(n)h(n) = heuristic estimate from node n to the goal
* h(n′)h(n') = heuristic estimate from successor n' to the goal
* c(n,n′)c(n, n') = actual cost of moving from n to n'

**Additionally:**  
Every consistent heuristic is **also admissible** — but not every admissible heuristic is consistent.

**📌 Example of a Consistent Heuristic**

**In pathfinding (like A\* search for maps):**  
If h(n) is the straight-line (Euclidean) distance from n to the goal, it’s consistent because:

* Moving from one location to a neighboring location cannot reduce the estimated distance to the goal by more than the cost to move to that neighbor.

This follows the **triangle inequality** from geometry:

distance from A to C≤distance from A to B+distance from B to C\text{distance from A to C} \leq \text{distance from A to B} + \text{distance from B to C}

**📌 Example: Admissible but Not Consistent Heuristic**

It’s rare but possible for a heuristic to be **admissible** (never overestimating the true cost to the goal) but **not consistent** (violating the monotonic property).

**Example Scenario: 8-Puzzle Game**

Suppose we define a heuristic:

* h(n) = 0 if n is the goal state
* h(n) = 2 for all other states

This heuristic is **admissible** because it never overestimates the true cost to the goal (since the actual minimum number of moves to solve from any state is ≥ 0).

**But it’s not consistent** because:

* If a state n has h(n) = 2 and one of its successors n' is the goal with h(n') = 0, and the move cost c(n, n') = 1, then:

2≰1+02 \nleq 1 + 0

which is false (since 2 > 1).  
So the heuristic fails the consistency condition.

**Task 4: Differences Between Tree Search and Graph Search**

**Objective:**

* Research the difference between **tree search** and **graph search** in the context of search algorithms.
* What are the advantages and limitations of each?
* When would each be preferred?

**Deliverable:**  
Comparison table + written explanation.

**📑 Tree Search vs Graph Search in AI Search Algorithms**

In AI, search algorithms are used to explore possible paths through a problem space to find a goal state. **Tree Search** and **Graph Search** are two fundamental strategies for navigating this search space — and while they may sound similar, they differ in how they handle repeated states and their search efficiency.

**📊 Comparison Table: Tree Search vs Graph Search**

| **Aspect** | **Tree Search** | **Graph Search** |
| --- | --- | --- |
| **Definition** | Explores the search space as a tree, expanding new nodes without checking for repeated states | Explores the search space while keeping track of visited (explored) states to avoid revisiting them |
| **Repeated States** | May revisit the same state multiple times, leading to redundant work and potential infinite loops | Avoids revisiting the same state by maintaining a record of explored states (often using a *closed list*) |
| **Memory Requirement** | Lower — only tracks nodes currently on the search path | Higher — must store explored states in memory |
| **Completeness** | May not be complete if cycles exist or if search space is infinite | Complete for finite search spaces (guaranteed to find a solution if one exists) |
| **Optimality (with A\*)** | Can fail to find optimal solution due to redundant paths | Guarantees optimal solution if using admissible heuristic |
| **Advantages** | Simpler to implement, uses less memory in small or non-repetitive spaces | More efficient in large, repetitive, or cyclic search spaces |
| **Limitations** | Inefficient in cyclic or large spaces, risk of infinite loops | Higher memory overhead due to storing explored states |
| **When to Use** | When the state space is small, acyclic, or memory is severely constrained | When the state space is large, contains cycles, or when avoiding repeated work is crucial |

**📌 Written Explanation**

**📍 Tree Search**

A **tree search** algorithm explores a search space by treating it like a tree, where each node represents a unique sequence of actions from the start state. It **does not remember previously explored states**, which makes it simple to implement but risky in problems with cycles or large state spaces.

**Example:**  
Depth-First Search (DFS) and Breadth-First Search (BFS) implemented as tree search can revisit the same state via different paths, causing inefficiency.

**Advantages:**

* Lower memory usage (no need for a closed list)
* Easier to implement for simple or small problems

**Limitations:**

* Can revisit the same state multiple times
* Not guaranteed to be complete or optimal in infinite or cyclic spaces

**Use When:**

* The problem has a small, acyclic search space
* Memory is limited
* The cost of revisiting states is negligible

**📍 Graph Search**

A **graph search** algorithm also explores the search space, but it **keeps track of explored states in a closed list** to avoid redundant work and prevent cycles. This makes it more reliable and efficient for larger, more complex search spaces.

**Example:**  
A\*, BFS, and DFS can be implemented as graph searches by maintaining an explored set.

**Advantages:**

* Avoids revisiting the same state
* Complete and optimal (when combined with admissible heuristics like in A\*)
* More efficient in large or cyclic spaces

**Limitations:**

* Requires more memory to store the explored set
* Slightly more complex implementation

**Use When:**

* The problem has a large or cyclic search space
* Optimality and completeness are important
* Memory is available to track visited states

**🧭 Level 2: Exploring Specific Algorithms**

**Task 5: Greedy Best-First Search (GBFS)**

**Objective:**

* Research how GBFS works.
* How does it select the next node to expand?
* What are its strengths and weaknesses?

**Deliverable:**  
Summary report + diagram of node selection.

Great — let’s build a clear, well-researched summary for **Task 5: Greedy Best-First Search (GBFS)**, including a diagram-style node selection explanation.

**📑 Task 5: Greedy Best-First Search (GBFS)**

**📌 What is Greedy Best-First Search?**

**Greedy Best-First Search (GBFS)** is an **informed search algorithm** that expands the node which appears to be **closest to the goal** based on a heuristic estimate. It focuses entirely on the estimated cost to reach the goal from the current node, **ignoring the path cost so far**.

It’s called “greedy” because it makes decisions based only on immediate benefit (the heuristic value), hoping to reach the goal quickly.

**📌 How Does GBFS Select the Next Node to Expand?**

At every step:

* GBFS selects the node with the **lowest heuristic value h(n)** from the open list (priority queue).
* It does **not consider the actual cost incurred so far (g(n))**.
* The heuristic function h(n) estimates the cost from node n to the goal.

**Selection Rule:**

Pick the node with the smallest h(n)\text{Pick the node with the smallest } h(n)

**📊 Diagram: GBFS Node Selection Process**

Let’s visualize this with a simple example:

**Graph:**

[A]

/ \

2/ \3

[B] [C]

/ \ \

1/ \4 \2

[D] [E] [F]

\

\1

[G] (Goal)

**Heuristic values (h):**

* h(A) = 6
* h(B) = 4
* h(C) = 5
* h(D) = 7
* h(E) = 2
* h(F) = 3
* h(G) = 0

**GBFS Process:**

1. Start at **A** (h=6)
2. Add **B(h=4)** and **C(h=5)** to open list
3. Pick **B** (lowest h)
4. Add **D(h=7)** and **E(h=2)**
5. Pick **E**
6. Add **G(h=0)** to open list
7. Pick **G** → Goal found

**Node selection order:**  
A → B → E → G

**📌 Strengths and Weaknesses of GBFS**

| **Strengths** | **Weaknesses** |
| --- | --- |
| Simple and easy to implement | Not guaranteed to find the optimal path |
| Can be faster than uninformed search methods | Can get stuck in loops or dead-ends without backtracking |
| Uses heuristic to guide search towards goal quickly | Completeness and optimality not guaranteed |
| Requires less memory compared to algorithms like A\* | Can ignore promising paths with slightly higher heuristic values |

***Task 6: A Search Algorithm*\***

**Objective:**

* Explain how A\* combines g(n) and h(n) to compute f(n).
* What is the role of each component?
* In what scenarios is A\* considered optimal?

**Deliverable:**  
Written explanation + one example problem (e.g., path in a map) with annotated steps (no coding).

**📑 A\* Search Algorithm**

**A\*** is one of the most popular and powerful **informed search algorithms**. It uses both:

* The **actual cost so far** to reach a node (g(n))
* And an **estimated cost to the goal** from that node (h(n))

to decide which node to expand next.

**📌 How A\* Combines g(n) and h(n)**

For each node n, A\* computes:

f(n)=g(n)+h(n)f(n) = g(n) + h(n)

Where:

* g(n) = actual cost from the start node to the current node n
* h(n) = heuristic estimate of the cost from n to the goal
* f(n) = estimated total cost of the cheapest solution through n

The node with the **lowest f(n) value** is selected for expansion next.

**📌 Role of Each Component**

| **Component** | **Role** |
| --- | --- |
| **g(n)** | Keeps track of the actual cost incurred to reach the current node from the start |
| **h(n)** | Provides an estimate of the cost from the current node to the goal, guiding the search efficiently |
| **f(n)** | Represents the total estimated cost of a path passing through n — used to prioritize node expansion |

This balanced combination makes A\* both **complete** and **optimally efficient** under certain conditions.

**📌 When is A\* Optimal?**

A\* is **guaranteed to find the optimal solution** when:

* The heuristic function h(n) is **admissible** (never overestimates the true cost to reach the goal)
* And ideally **consistent (monotonic)** (satisfies: h(n)≤c(n,n′)+h(n′)h(n) \leq c(n, n') + h(n'))

When these conditions hold:

* A\* expands the least number of possible nodes necessary to guarantee finding the shortest path.

**📊 Example Problem: Pathfinding on a Map**

**Goal:** Find the shortest path from point **A** to **G**

**Graph:**

A

/ \

(1) (4)

B C

| \ |

(3)(1) (2)

D E F

\ | /

(6)(1)(3)

G

**Heuristic (h) values:**

* h(A) = 7
* h(B) = 6
* h(C) = 5
* h(D) = 4
* h(E) = 2
* h(F) = 1
* h(G) = 0

**📌 A\* Step-by-Step (No Coding, Annotated)**

| **Step** | **Current Node** | **Open List (f(n))** | **Chosen Node** |
| --- | --- | --- | --- |
| 1 | A | B(1+6=7), C(4+5=9) | B |
| 2 | B | C(9), D(4+4=8), E(2+2=4) | E |
| 3 | E | C(9), D(8), G(3+0=3) | G (Goal reached) |

**Path Found:**  
A → B → E → G

**Actual Cost (g):**  
1 (A to B) + 1 (B to E) + 1 (E to G) = **3**

**📌 Why This Worked:**

* At each step, A\* prioritized the node with the **lowest f(n) value**
* Balanced between **actual cost so far (g)** and **estimated remaining cost (h)**
* Because h(n) was admissible and consistent, A\* efficiently found the optimal path.

***Task 7: Why A is Optimal (with Admissible Heuristic)*\***

**Objective:**

* Explore the reasoning or proof sketch of why A\* is guaranteed to find an optimal path if h(n) is admissible.
* Why is consistency even better?

**Deliverable:**  
Write-up in bullet points + cited source.

**📑 Why A\* is Optimal (with Admissible Heuristic)**

**📌 Reasoning and Proof Sketch (in Bullet Points)**

* ✅ **A\* Search selects the node with the lowest f(n) = g(n) + h(n)** from the open list.
* ✅ If the heuristic h(n) is **admissible**, it **never overestimates the true cost** to reach the goal from any node.
* ✅ This ensures that the **first time the goal node is selected for expansion**, it has the **lowest possible total path cost**.
* ✅ No other path to the goal can have a lower f(n) than the one already selected, because:
  + Any alternative path would either:
    - Already be in the open list with a higher or equal f(n)
    - Or have been expanded already with a higher g(n)
* ✅ As a result, A\* is **complete** (it finds a solution if one exists) and **optimal** (the solution has the lowest possible cost).

**📌 Why is Consistency Even Better?**

* ✅ A **consistent (monotonic) heuristic** satisfies:

h(n)≤c(n,n′)+h(n′)h(n) \leq c(n, n') + h(n')

for every node n and its successor n'.

* ✅ This guarantees that the **f-values along any path are non-decreasing** — meaning once a node is expanded in A\*, it **never needs to be revisited**.
* ✅ This improves efficiency because:
  + It eliminates the need to check or update already expanded nodes.
  + It makes the open and closed lists management simpler.
  + It preserves optimality just like an admissible heuristic, but with less overhead.

**📌 Summary of Key Points**

* **Admissibility ensures optimality** because it prevents A\* from overlooking a cheaper path by guaranteeing heuristic estimates are optimistic.
* **Consistency improves efficiency** by preventing re-expansions of nodes and ensuring f(n) values along any path are non-decreasing.
* **If the heuristic is consistent, it is automatically admissible.**

**📌 Cited Source**

**Primary Reference:**  
Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.  
📖 See Chapter 3 — *Solving Problems by Searching*, especially section on A\* and properties of heuristics.

**Task 8: Comparison Table – BFS, DFS, GBFS, A\***

**Objective:**  
Create a table comparing the following algorithms:

* Breadth-First Search
* Depth-First Search
* Greedy Best-First Search
* A\* Search

**Compare on:**

* Use of heuristic
* Completeness
* Optimality
* Time complexity
* Space complexity

**Deliverable:**  
Clean comparison table.

**📊 Comparison Table: BFS, DFS, GBFS, A\***

| **Feature** | **Breadth-First Search (BFS)** | **Depth-First Search (DFS)** | **Greedy Best-First Search (GBFS)** | **A\* Search** |
| --- | --- | --- | --- | --- |
| **Use of Heuristic** | ❌ No | ❌ No | ✅ Yes (h(n)) | ✅ Yes (g(n) + h(n)) |
| **Completeness** | ✅ Yes (if finite branching factor and finite goal depth) | ❌ No (may get stuck in infinite path) | ❌ No (may get stuck in loops or dead ends) | ✅ Yes (if h(n) is admissible and positive cost) |
| **Optimality** | ✅ Yes (for uniform step cost) | ❌ No | ❌ No | ✅ Yes (if h(n) is admissible) |
| **Time Complexity** | O(bd)O(b^d) | O(bm)O(b^m) | O(bm)O(b^m) (worst-case, with poor heuristic) | O(bd)O(b^d) (with good heuristic, depends on h(n)) |
| **Space Complexity** | O(bd)O(b^d) | O(bm)O(bm) | O(bm)O(b^m) | O(bd)O(b^d) |
| **Best Use Case** | Finding shortest path in small, uniform-cost spaces | Deep searches with memory constraints | Fast approximate searches with good heuristic | Finding optimal path efficiently with admissible heuristic |

**Legend:**

* bb = branching factor (average number of successors per state)
* dd = depth of the shallowest solution
* mm = maximum depth of the search tree

**🌍 Level 3: Real-World and Heuristic Design**

**Task 9: Heuristics in Real Life**

**Objective:**  
Find examples of heuristics used in **real-world applications**:

* GPS navigation
* Game AI (e.g., Chess, Pacman)
* Robotics

**Deliverable:**  
One-slide explanation for each real-world example (include images if possible).

**📑 Task 9: Heuristics in Real-World Applications**

**📍 Slide 1: GPS Navigation (Pathfinding)**

🗺️ **Application:**  
GPS systems (like Google Maps, Waze) use heuristics to suggest the fastest or shortest route to a destination.

🎯 **Heuristic Example:**  
**Straight-line (Euclidean) distance** from current location to destination is used as a heuristic estimate of remaining travel.

📌 **How it works:**

* The system estimates how far a location is "as the crow flies."
* Prioritizes roads leading in that direction, factoring in road type, speed limits, and traffic.

📈 **Why it's effective:**  
It quickly eliminates unlikely routes and focuses on the most promising paths to compute the shortest or fastest travel time.

**Image Idea:**

* A map with routes and a highlighted straight line from point A to B labeled “Heuristic estimate”.

**📍 Slide 2: Game AI (Chess / Pacman)**

🎮 **Application:**  
Game AIs use heuristics to make decisions about moves without evaluating every possible future outcome.

🎯 **Heuristic Example (Chess):**  
**Material advantage heuristic** assigns point values to each piece:

* Queen = 9, Rook = 5, Bishop = 3, Knight = 3, Pawn = 1.

📌 **How it works:**

* The AI evaluates possible moves by summing the value of remaining pieces.
* Picks moves that improve its position according to this score.

🎯 **Heuristic Example (Pacman):**  
**Manhattan distance** between Pacman and the nearest dot/ghost to decide movement.

📈 **Why it's effective:**  
It allows fast, strategic decisions without exhaustive computation.

**Image Idea:**

* Chessboard with point values next to each piece
* Pacman map with dots and ghosts, and arrows showing distances.

**📍 Slide 3: Robotics (Autonomous Navigation)**

🤖 **Application:**  
Robots use heuristics for real-time pathfinding and obstacle avoidance in physical environments.

🎯 **Heuristic Example:**  
**Estimated distance to goal (Euclidean or Manhattan)** combined with sensor data for nearby obstacles.

📌 **How it works:**

* Robot uses sensors (LIDAR, sonar) to map surroundings.
* Heuristic estimates cost of moving toward goal while avoiding obstacles.

📈 **Why it's effective:**  
Balances path efficiency with safety by avoiding areas with high risk or delay.

**Image Idea:**

* A robot navigating a grid with obstacles
* Path marked based on estimated costs, with goal location highlighted.

**Task 10: Designing a Heuristic Function (Conceptual)**

**Objective:**  
Choose one of the following:

* Maze solving
* Food delivery path planning
* Robot vacuum cleaning

Design a heuristic *conceptually*:

* What would you estimate as the remaining cost to goal?
* What factors would your heuristic consider?

Great — let’s pick **Food Delivery Path Planning** for Task 10 and conceptually design a heuristic for it. I’ll structure it neatly for your submission:

**📑 Task 10: Designing a Heuristic Function — Food Delivery Path Planning**

**📌 📦 Problem Context:**

An AI system plans the most efficient path for a food delivery agent to reach a customer's location through city streets, avoiding traffic and delays.

**📌 🎯 What Would You Estimate as the Remaining Cost to Goal?**

The heuristic would estimate the **remaining delivery time** from the current location to the customer's address.

h(n)=estimated time (in minutes) to reach the customer from node nh(n) = \text{estimated time (in minutes) to reach the customer from node } n

**📌 📊 Factors the Heuristic Would Consider:**

| **Factor** | **Why It Matters** |
| --- | --- |
| **Straight-line distance to goal** | Basic estimate of how far the location is geographically |
| **Current traffic conditions** | Affects actual travel time — higher traffic, longer travel |
| **Road types and speed limits** | Highways and major roads allow faster travel than small streets |
| **Number of traffic signals and junctions** | Each stop or turn can increase delivery time |
| **Weather conditions (optional)** | Heavy rain or bad weather slows down movement |
| **Ongoing road closures or diversions** | May force detours, increasing estimated travel time |

**📌 📈 Conceptual Heuristic Formula:**

A possible simplified conceptual formula:

h(n)=d(n,goal)vavg(n)+traffic\_delay(n)h(n) = \frac{d(n, \text{goal})}{v\_{\text{avg}}(n)} + \text{traffic\\_delay}(n)

Where:

* d(n,goal)d(n, \text{goal}) = straight-line distance to goal
* vavg(n)v\_{\text{avg}}(n) = average effective speed on roads from n (based on road type and traffic)
* traffic\_delay(n)\text{traffic\\_delay}(n) = estimated time lost in traffic signals, jams, or diversions

**📌 ✅ Why This Heuristic Would Be Effective:**

* **Admissible:** It estimates real travel time based on distance and traffic but won't overestimate.
* **Informed:** Uses real-world factors (traffic, speed limits) for accurate route guidance.
* **Efficient:** Helps prioritize faster, less congested routes over shorter but slower ones.