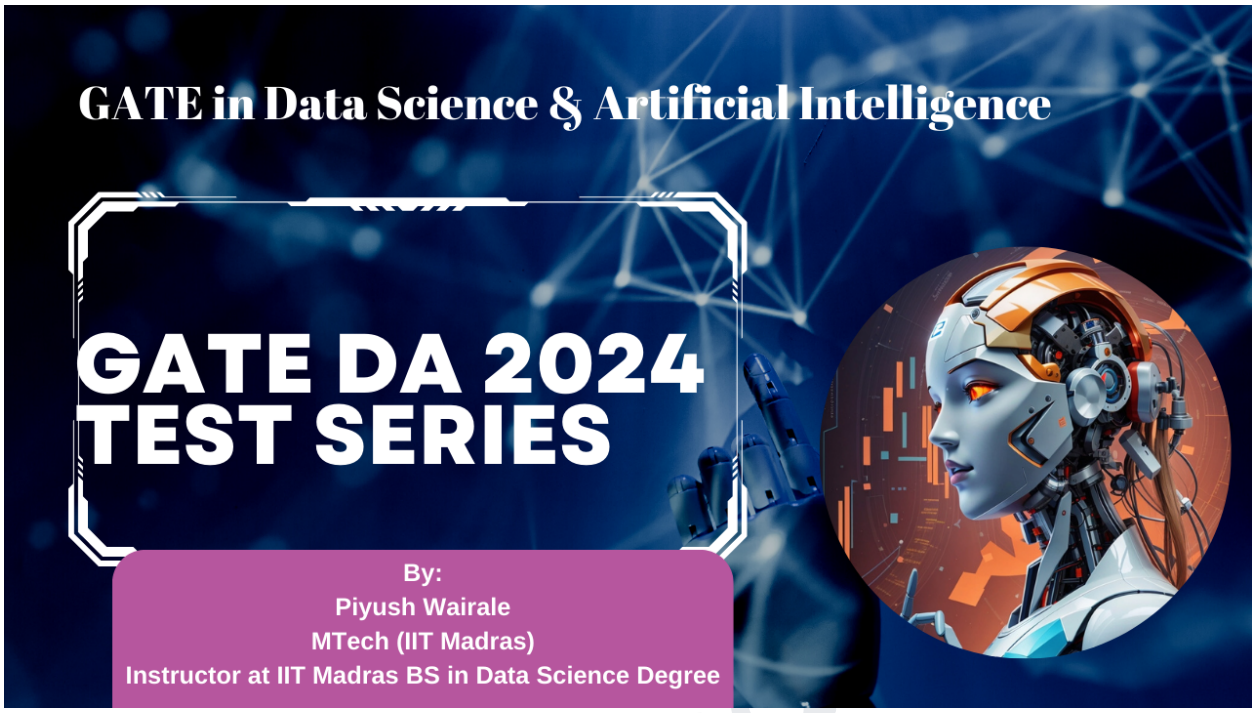


Connect with me:
Youtube: Piyush Wairale IITM
Linkedin: Piyush Wairale
Facebook: Piyush Wairale
Instagram: Piyush Wairale

2. Informed Search:

Breadth-first and depth-first are both uninformed search algorithms. That is, these algorithms do not utilize any knowledge about the problem that they did not acquire through their own exploration. However, most often is the case that some knowledge about the problem is, in fact, available. For example, when a human maze-solver enters a junction, the human can see which way goes in the general direction of the solution and which way does not. AI can do the same. A type of algorithm that considers additional knowledge to try to improve its performance is called an informed search algorithm.

1. Best-first search is an informed search strategy that selects nodes for expansion based on an evaluation function, f(n), with the lowest evaluation being expanded first.
2. The evaluation function in best-first search is often a cost estimate, and it determines the search strategy. It can include a heuristic function, h(n), which estimates the cost to reach a goal state from the current node.
3. Heuristic functions are a common way to provide problem-specific knowledge to the search algorithm, and they are typically nonnegative and state-dependent.
4. Heuristic functions must satisfy the constraint that if a node is a goal state, its heuristic value is 0.
5. Best-first search, guided by heuristic information, can significantly improve search efficiency compared to uninformed search strategies and can be adapted to various problem domains.

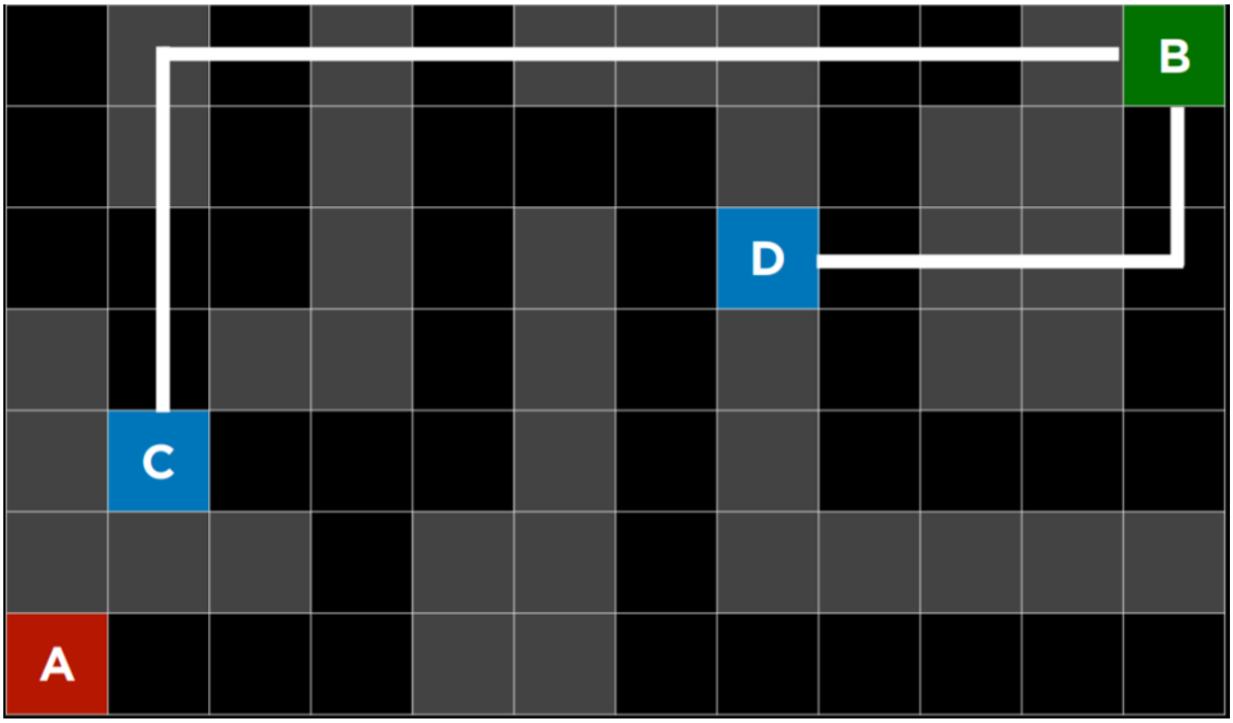
Type of Informed Search:

1. Best First Search Algorithm(Greedy search)
2. A* Search Algorithm

2.1 Best First Search Algorithm (Greedy Search):

Greedy best-first search expands the node that is the closest to the goal, as determined by a heuristic function *h(n)*. As its name suggests, the function estimates how close to the goal the next node is, but it can be mistaken. The efficiency of the *greedy best-first* algorithm depends on how good the heuristic function is. For example, in a maze, an algorithm can use a heuristic function that relies on the Manhattan distance between the possible nodes and the end of the maze. The *Manhattan distance* ignores walls and

counts how many steps up, down, or to the sides it would take to get from one location to the goal location. This is an easy estimation that can be derived based on the (x, y) coordinates of the current location and the goal location.



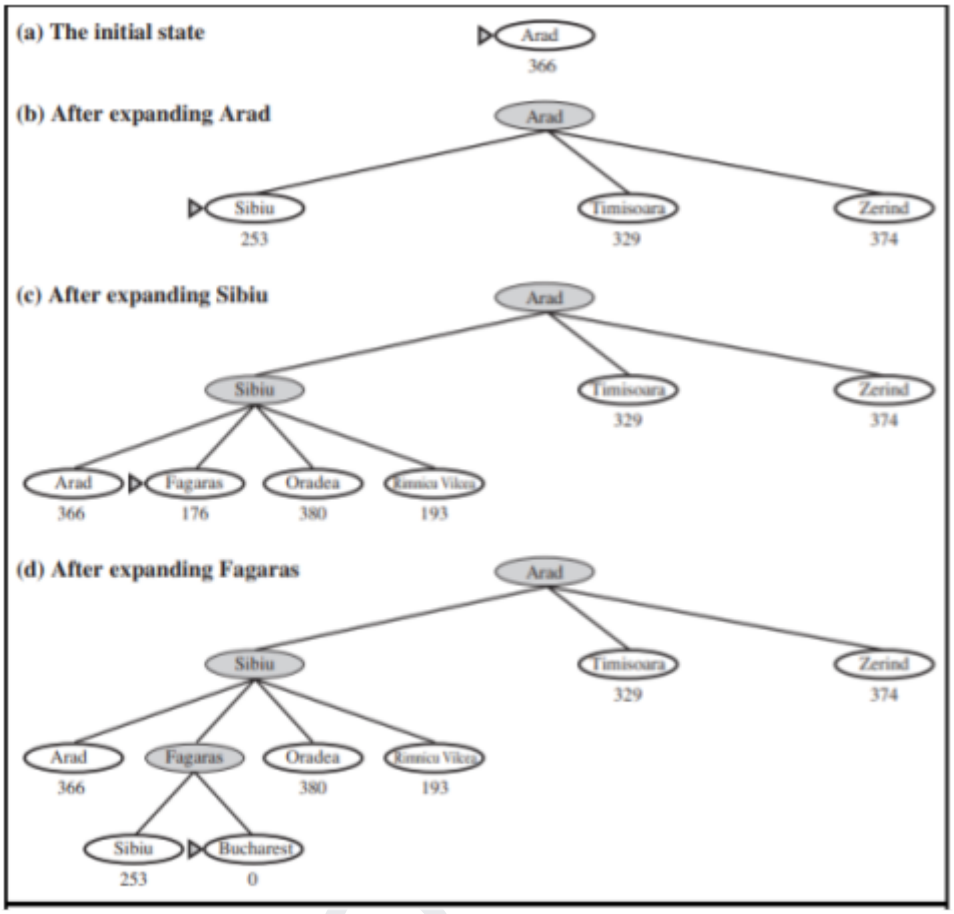
(Kindly refer to lectures for detailed explanation of above example)

Manhattan Distance

However, it is important to emphasize that, as with any heuristic, it can go wrong and lead the algorithm down a slower path than it would have gone otherwise. It is possible that an *uninformed* search algorithm will provide a better solution faster, but it is less likely to do so than an *informed* algorithm.

1. Greedy best-first search relies solely on the heuristic function (f(n) = h(n)) to evaluate nodes and selects the one closest to the goal, aiming for a quick solution.
2. In the context of route-finding problems in Romania, the straight-line distance heuristic (hSLD) estimates distances to Bucharest, aiding the search algorithm.
3. Heuristic values, like hSLD, cannot be deduced from the problem description alone and may require domain-specific knowledge.
4. Greedy best-first search is not optimal and may not always find the shortest path, as it prioritizes nodes that seem closest to the goal.
5. In some cases, greedy best-first search can become incomplete, leading to infinite loops when dealing with certain state spaces, such as the Iasi to Fagaras problem.

Best First Search is an informed search algorithm that selects the next node to expand based on a heuristic evaluation of its potential to reach the goal quickly.



Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic hSLD . Nodes are labeled with their h-values.

Algorithm of Best First Search (Greedy Search):

1. Start with an initial node containing the starting state of the problem and a path cost of 0.
2. Check if the initial state is a goal state. If it is, you're done, and you have a solution.
3. Create a priority queue called the "frontier" and add the initial node to it, using the heuristic function to prioritize nodes.
4. Create an empty set called "explored" to keep track of the states we've already seen.
5. Repeat the following steps:
    a. If the frontier is empty (there's no one left in the queue) and you haven't found a solution, then you've failed to find a solution. So, return failure.

   b. Take the node with the highest heuristic value from the frontier (the most promising one).
   c. Add the state of this node to the explored set, so we remember we've seen it.
   d. For each possible action you can take from the current state:
      i. Create a new node that represents the result of taking that action.
      ii. Calculate the heuristic value for the new node. iii. If the new node's state is not in the explored set or the frontier, add it to the frontier.
6. Keep repeating these steps until you either find a solution or determine that there is no solution (if the frontier becomes empty).

Best First Search (Greedy Search) Performance: Now, let's evaluate the performance of the Be**st First Search algorithm:**

- **Completeness:** Best First Search is not guaranteed to be complete. It may get stuck in loops or explore paths that lead to dead ends, especially if the heuristic is not admissible.
- **Optimality:** Best First Search is not guaranteed to find the optimal solution. It tends to prioritize nodes that appear to be closer to the goal based on the heuristic, which may not always lead to the shortest path.
- **Time Complexity:** The time complexity depends on the quality of the heuristic function. In the worst case, it can be exponential if the heuristic is not admissible. However, with a good heuristic, it can be very efficient.
- **Space Complexity:** The space complexity depends on the size of the frontier, which can vary. In the worst case, it can be exponential, but with a good heuristic, it can be much smaller.

Advantages of Best First Search (Greedy Search):

- **Efficiency:** Best First Search can be very efficient with a well-designed heuristic, quickly converging to a solution.
- **Heuristic Guidance:** It leverages heuristic information to prioritize nodes that seem promising, which can be helpful in finding solutions faster.

Disadvantages of Best First Search (Greedy Search):

- **Lack of Completeness:** Best First Search is not guaranteed to find a solution, and it can get stuck in infinite loops.
- **Lack of Optimality:** It may find a solution, but it might not be the optimal one, as it tends to follow the heuristic's guidance.

2.2 A* Search Algorithm:

A development of the *greedy best-first* algorithm, *A* search* considers not only *h(n)*, the estimated cost from the current location to the goal, but also *g(n)*, the cost that was accrued until the current location. By combining both these values, the algorithm has a more accurate way of determining the cost of the solution and optimizing its choices on the go. The algorithm keeps track of (*cost of path until now + estimated cost to the goal*), and once it exceeds the estimated cost of some previous option, the algorithm will ditch the current path and go back to the previous option, thus preventing itself from going down a long, inefficient path that *h(n)* erroneously marked as best.

Yet again, since this algorithm, too, relies on a heuristic, it is as good as the heuristic that it employs. It is possible that in some situations it will be less efficient than *greedy best-first* search or even the *uninformed* algorithms.

1. A∗ search combines the cost to reach a node (g(n)) and the estimated cost to reach the goal from that node (h(n)) to evaluate nodes, using the formula f(n) = g(n) + h(n).
2. The value of f(n) represents the estimated cost of the cheapest solution passing through node n.
3. A∗ search aims to find the cheapest solution by prioritizing nodes with the lowest f(n) values.
4. A∗ search is both complete and optimal under certain conditions for the heuristic function h(n).
5. While A∗ search is similar to uniform-cost search, it uses the sum of g and h values instead of just g.

Conditions for optimality: Admissibility and consistency:

For *A* search* to be optimal, the heuristic function, *h(n)*, should be:

1. *Admissible*, or never *overestimating* the true cost, and
2. *Consistent*, which means that the estimated path cost to the goal of a new node in addition to the cost of transitioning to it from the previous node is greater or equal to the estimated path cost to the goal of the previous node. To put it in an equation form, *h(n)* is consistent if for every node *n* and successor node *n'* with step cost *c*, $h(n) \leq h(n') + c$.

- An **admissible heuristic** in A* search is one that never overestimates the cost to reach the goal. This means that the estimated cost to reach the goal from any node (h(n)) is always less than or equal to the true cost. Admissible heuristics guide the A* algorithm to explore nodes efficiently, ensuring it searches for the most cost-effective solutions.
- **Consistency,** also known as **monotonicity**, is a slightly stricter condition applicable to A∗ in graph search. A heuristic h(n) is consistent if, for any node n and its successor n', generated by an action a, the estimated cost of reaching the goal from n is less than or equal to the true cost of reaching n' plus the estimated cost of reaching the goal from n'. This relationship is expressed as: h(n) ≤ cost of getting to n' from n + h(n').
- The **triangle inequality** conceptually resembles this property: the direct path from n to the goal is no longer than the sum of two shorter paths from n to n' and from n' to the goal.
- These properties are essential for A∗ search to provide both completeness and optimality while evaluating nodes based on the function f(n) = g(n) + h(n).

Optimality of A*

★ As we mentioned earlier, A∗ has the following properties: *the tree-search version of A∗ is optimal if h(n) is admissible, while the graph-search version is optimal if h(n) is consistent.*

★ The first step is to establish the following: if *h(n) is consistent, then the values of f(n) along any path are nondecreasing*. Suppose n is a successor of n; then g(n') = g(n) + c(n, a, n') for some action a, and we have,

$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$.

★ The next step is to prove that whenever *A∗ selects a node n for expansion, the optimal path to that node has been found.*

- ➔ **Contours** in the state space represent bands of nodes with increasing f-costs. A* expands nodes with f(n) less than the cost of the optimal solution path (C*). If C* is the cost of the optimal solution path, then we can say the following:
  • A* expands all nodes with f(n) < C*.
  • A* might then expand some of the nodes right on the "goal contour" (where f(n) = C*) before selecting a goal node
- ➔ A* search **prunes** subtrees when using an admissible heuristic, allowing it to avoid exploring certain paths while still guaranteeing optimality.
- ➔ A* is **optimally efficient** among algorithms that extend search paths from the root and use consistent heuristics, ensuring that no other optimal algorithm is guaranteed to expand fewer nodes.
- ➔ The problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the **absolute error** or the **relative error** of the heuristic. The absolute error is defined as $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal, and the relative error is defined as $\equiv (h^* - h)/h^*$.

Algorithm of A* Search:

1. Start with an initial node containing the starting state of the problem and a path cost of 0.
2. Check if the initial state is a goal state. If it is, you're done, and you have a solution.
3. Create a priority queue called the "frontier" and add the initial node to it, using the sum of the path cost and a heuristic function's estimate to prioritize nodes.
4. Create an empty set called "explored" to keep track of the states we've already seen.
5. Repeat the following steps:
   a If the frontier is empty (there's no one left in the queue) and you haven't found a solution, then you've failed to find a solution. So, return failure.
   b. Take the node with the lowest combined cost (path cost + heuristic estimate) from the frontier,
   c. Add the state of this node to the explored set, so we remember we've seen it.
   d. For each possible action you can take from the current state:
   i. Create a new node that represents the result of taking that action.
   ii. Calculate the path cost from the initial state to this new node.
   iii. Calculate the heuristic estimate from this new node to the goal.
   iv. Calculate the combined cost (path cost + heuristic estimate).
   v. If the new node's state is not in the explored set or the frontier, or if it's in the frontier but with a higher combined cost, update its path cost and add it to the frontier.

6. Keep repeating these steps until you either find a solution or determine that there is no solution (if the frontier becomes empty).

A* Search Algorithm Performance:

Now, let's evaluate the performance of the A* Search algorithm:

- **Completeness**: A* Search is complete if both the heuristic function and the cost function satisfy certain conditions. It will find a solution if one exists, provided these conditions are met.
- **Optimality:** A* Search is guaranteed to find the optimal solution as long as the heuristic function is admissible (never overestimates the true cost) and consistent.
- **Time Complexity**: The time complexity depends on the quality of the heuristic function. In the worst case, it can be exponential if the heuristic is not admissible. However, with a good heuristic, it is typically very efficient.
- **Space Complexity:** The space complexity depends on the size of the frontier, similar to other search algorithms. In the worst case, it can be exponential, but with a good heuristic, it can be much smaller.

Advantages of A* Search:

- **Completeness and Optimality**: A* Search combines completeness and optimality, making it a reliable choice for finding the shortest path.
- **Heuristic Guidance:** It leverages heuristic information to prioritize nodes efficiently.
- **Adaptable:** A* Search can be tailored to different problems by adjusting the heuristic function.

Disadvantages of A Search:

- **Heuristic Requirement:** A* Search requires a well-designed heuristic function to perform optimally.
- **Memory Usage:** Like other search algorithms, A* Search may require a significant amount of memory in some cases, especially with deep search trees.

👉 *The performance of heuristic search algorithms depends on the quality of the heuristic function. One can sometimes construct good heuristics by relaxing the problem definition, by storing precomputed solution costs for subproblems in a pattern database, or by learning from experience with the problem class.*

References:

- FIRST **COURSE** IN **ARTIFICIAL INTELLIGENCE**. **Deepak** Khemani.
- CS50: Introduction to AI course, Harvard University
- Artificial Intelligence: A Modern Approach Textbook by Peter Norvig and Stuart J. Russell
- Artificial Intelligence Book by Saroj Kaushik
- IIT Madras BS in Data Science Degree Lectures